Tampere University

Riku Lehto

LOAD BALANCING DISTRIBUTED COMPUTATION

Master of Science thesis
Information Technology
Examiners: Professor Hannu-Matti Järvinen and
Master of Science Vesa Sarviluoma
April 2020

ABSTRACT

RIKU LEHTO: Load balancing distributed computation
Tampere University
Master of Science thesis, 45 pages
April 2020
Master's Degree Programme in Information Technology
Major: Pervasive Systems
Examiners: Professor Hannu-Matti Järvinen and Master of Science Vesa Sarviluoma
Keywords: load balancing, distributed computation

Distributed computing is currently a popular way to harness great amounts of processing power by combining multiple computation resources together and distributing workloads over them. This way, the system is capable of providing much more processing power than any one computation resource can. By distributing work over multiple hardware components the system also gains fault tolerance, responsiveness and scalability. This thesis focuses on a system, which consists of multiple computation platforms with very little computational power on a local network, and how to schedule workloads to each of them.

We design a load balancer to handle requests for computational resources and computation client software capable of harnessing these resources. We build a computation server which also runs the computation client software with multiple times the computation power of one regular computation platform.

Multiple load balancing methods are designed for the load balancer to use and benchmarked to find the most performant one. We also benchmark the performance difference between a regular computation client and the computation server client and estimate how many regular clients one server can replace. The results show that one computation server can replace about 15 to 30 computation clients and that a simple scheduling algorithm with an index of where the scheduling was left off beats schedulers with more complicated logic. We also propose that further research should be done about the responsiveness of the algorithms.

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

TIIVISTELMÄ

RIKU LEHTO: Hajautetun laskennan kuormantasaus
Tampereen yliopisto
Diplomityö, 45 sivua
Huhtikuu 2020
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Pervasive Systems
Tarkastajat: professori Hannu-Matti Järvinen ja diplomi-insinööri Vesa Sarviluoma
Avainsanat: kuormantasaus, hajautettu laskenta

Hajautettu laskenta on nykyään suosittu tapa valjastaa suuria määriä laskentatehoa yhdistämällä useita laskentaresursseja yhteen ja jakamalla työtaakan niiden välillä. Täten järjestelmä pystyy tarjoamaan paljon enemmän laskentatehoa kuin mikään yksittäinen laskentaresurssi itsenään. Hajauttamalla työtä monen rautakomponentin välillä järjestelmän vikasietoisuus, reagointikyky ja skaalautuvuus paranevat. Tämä diplomityö keskittyy järjestelmään, joka koostuu useasta laskenta-alustasta, joiden laskentateho on hyvin matala, paikallisessa verkossa ja kuinka niiden välinen kuormantasaus tapahtuu.

Järjestelmään suunnitellaan kuormantasain ja asiakasohjelma, jonka tehtävänä on valjastaa laskenta-alustojen laskentateho järjestelmän käyttöön. Järjestelmän käyttöön suunnitellaan myös laskentapalvelin moninkertaisella laskentateholla verratain normaaliin laskenta-alustaan, jolla asiakasohjelmaa myös ajetaan.

Järjestelmää varten suunnitellaan useita kuormantasaus-algoritmeja, joiden suorituskykyä vertaillaan keskenään. Myöskin normaalin laskenta-alustan ja laskentapalvelimen suorituskykyä vertaillaan toisiinsa ja tästä arvioidaan kuinka monta laskenta-alustaa yksi laskentapalvelin pystyy korvaamaan. Tulosten perusteella voidaan sanoa, että laskentapalvelin pystyy korvaamaan noin 15-30 laskenta-alustaa, ja että yksinkertainen kuormantasaus-algoritmi osoittimella viimeisimpään kuormantasauspäätökseen on paljon tehokkaampi kuin algoritmit joissa on monimutkaisempaa logiikaa. Ehdotamme myös, että kuormantasaus-algoritmien reagointikykyä tulisi tutkia lisää.

PREFACE

This thesis was written between May 2019 and April 2020. The thesis is based on a project done for FLS Finland Oy while I was employed by Bitwise Oy. I am very grateful for the opportunity to work with the project in my thesis and enjoyed designing and benchmarking the resulting system.

I would like to thank all my colleagues who helped me create this thesis. Special thanks go to FLS Finland Oy for giving me the opportunity to base my thesis on the project I have worked so much on and enjoy working with.

Riku Lehto
Tampere, 28 April, 2020

Contents

1 INTRODUCTION

Distributed computing and IoT have been gaining significant importance in the modern world over the past few years. Inexpensive hardware with internet connection capabilities is scattered all over the world, reading sensor data, computing basic tasks and communicating with a larger, central server or even with each other over the network. These networks allow us to get almost live data from almost any location, at any time. Applications for these kinds of networks are endless: want to monitor your home from work? Buy a bunch of IoT cameras and connect them to your home server or buy a similar service from an IoT company. Want live usage data and heat maps of how people use public spaces? A few cameras, a neural network to detect people and good enough coverage will solve your problem. Weather reports from the middle of nowhere? Live diagnostics data from those hard to reach installations in the woods? If you can fit a little weather-protected chip somewhere, you can make it use IoT. But what do we do with all of this computing power and storage space all over when it is not actively processing or doing anything?

This thesis focuses on a system monitoring a parking area with several IoT boards. One board contains a selection of different sensors, from cameras to proximity and movement sensors, microphones, and temperature sensors. The boards also have different visual and audible outputs, mainly for user interaction. All the boards are connected to a central server, which compiles data to a near real-time visual representation of the monitored area, combines multiple events to more advanced event types and stores all of this to a database for searching and analysing. In the current state, each board runs its own gathered data through a detection algorithm pipeline locally before sending the processed data to the server. The server runs additional algorithms to create a complete state of the monitored area and stores it into a database for other applications to parse and visualize for the user.

This approach makes the system highly scalable, as the bulk of the computation is made on each unit, and the server only makes computations for the already processed data. The computational power of the units, however, is quite limited and would greatly benefit from hardware accelerated image processing and a dedicated GPU. The units can only run the detection algorithm for one detected car at a time, increasing the latency between multiple detections for one unit.

We came up with the idea of improving the performance of individual board using distributed computing by chance, as we know that most of the time the boards have a little to no computational load, but require huge amounts of parallel computational power in short bursts when the detection algorithms are run. This idea was tossed around for a while, and we decided to test it when tasked to improve the performance

of the detection algorithms. By creating a simple prototype and seeing if it works we get results fast and see if the idea is in any form feasible without needing to design complex models or designs for the system that may not even work.

New client software is implemented for the system, which will be run on all the units and a dedicated offloading server with server grade CPU and GPU. This "computation unit" software will receive image data from units monitoring parking spaces in the parking area and run the computationally intensive algorithms for them. This should enable us to run several detections at once in parallel, as we can distribute the work of one unit between multiple when the other units have free processing power. We propose and benchmark a few different load balancing methods to be used to schedule computation resources to be used by each board.

In Chapter 2, we specify the currently running system. Chapter 3 touches a bit on the theory behind distributed computing and load balancing. Chapter 4 describes the new "computing unit" software capable of running computation tasks over the network and a server hardware build used in the benchmarks. Chapter 5 lists all the changes needed to migrate from the current setup to the new one. Chapter 6 contains all the benchmarks, from the current performance of the algorithms to the scheduling methods. In Chapter 7, we draw some conclusions and plan what's next for this feature.

2 CURRENT SYSTEM

This chapter details the current server and the client, how they work, what workloads both of them run and how the two communicate. The first section details the client, the second section details the server and the final section the communication between the two.

2.1 Client

The client is a collection of multiple software components run on a dedicated hardware platform. The client software can be run on almost any hardware, but this thesis focuses on the dedicated board for its benchmarks. The following details the most relevant parts of the hardware, software and the image detection pipeline.

Hardware

The hardware platform the client is running on is a custom-built board with the following specifications.

CPU Model	ARM Cortex-A7
CPU Cores	2
CPU Clock speed	1.0 GHz
CPU Features	half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae evtstrm
RAM	1 GiB or 2 GiB
Network devices	Two 100/10 Ethernet ports
Power In	2 parallel 24V DC inputs
Video IO	2 MIPI camera serial interfaces (CSI)
Audio In	Extrenal microphone
Audio Out	External speaker
General IO	General-purpose input/output (GPIO)
Human Interface Devices (HIDs)	2 LED boards with 5 RGB LEDs each

Table 1: Hardware

Table 1 contains detailed information about the hardware platform the client software is run on. The information was gathered from the platform's data sheet.

The processor is an overclocked Arm Cortex-A7 [8] with two cores, enabling some level of parallel computation which is heavily utilized in image processing and machine learning algorithms. The processors NEON VFPV3 and VFPV4 [18] features are also helpful for image processing operations, contributing a significant boost in performance.

The board can have either 1 GiB or 2 GiB of RAM installed depending on the amount of traffic expected in the monitored area. More traffic means more activations for video recording and image processing which need more RAM to be available as a buffer.

The board has two Ethernet ports and two parallel DC inputs, making it possible to create chains of the devices over a large distance. This makes covering large monitorable areas easier, as each unit doesn't have to have cables running all the way back to the network switch and power supply, but only to the previous board.

Two MIPI camera serial interfaces (CSI) [26] provide for a simple way to interface with a camera mounted next to the board. The board is capable of capturing 2 frames per second for each camera or 4 frames per second on one camera depending on the configuration and attached cameras.

Audio devices on the board consist of an external microphone and speaker. These can be used to both capture and playback audio for different events.

The board has a set of General Purpose Input/Output (GPIO) pins to use with different external devices. These are used to read sensor data and control several different attached hardware components programmatically.

Two LED-controller boards are also attached to most boards, enabling the board to signal the nearby users of its state. The board can use these to display the state of the nearby parking spaces or signal an administrator with a special state, such as overtime parking or a fault in the unit.

Software

The board runs a custom embedded Linux operating system built with the Yocto project -toolchain [48]. The built image contains the basic functionalities needed for the operating system and networking. Additional packages include SSH-server for remote shell access and GStreamer [16] for interfacing with onboard camera drivers.

Several different software binaries are compiled and run on the board, creating a network of programs communicating with each other using UNIX sockets. Binaries

exist for reading sensor inputs, controlling hardware outputs and auxiliary devices, running image processing and communicating with the server. A single ”main” program sits in the middle of all this, directing the whole setup.

This main program is the main interest in this thesis as it runs the primary vehicle detection pipeline. It receives all the different sensor inputs from other programs and runs them through different detection algorithms. The program also connects to the server on startup, registers itself as an active board and receives a configuration if one is available. It maintains a connection with the server at all times, sending events on detections and other events to the server and responding to queries from the server, e.g. heartbeat messages or manual data request. The program also controls onboard LEDs and audio output to signal the nearby users of its state, for example the occupancy state of monitored parking spaces.

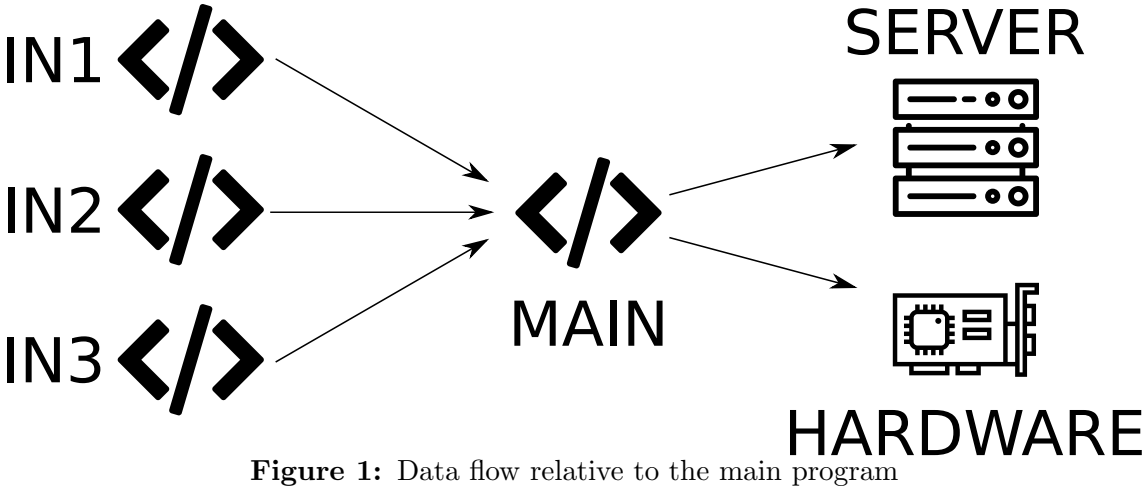

Figure 1: Data flow relative to the main program

Figure 1 depicts data flow relative to the main program. On the left three programs labelled IN1-3 feed input data to the main program. On the right the server and hardware receive input from the main program after it has processed the input data.

All the runnable tasks and algorithms are compiled into separate libraries. This is done to create a clear interface point for usage in different programs. This also speeds up compilation times significantly and saves storage space as different programs can share libraries between themselves.

The main program must also support an ”offline mode”, where it can keep functioning even if the connection to the server is lost. It must store all detection events until the connection to the server is re-established and it must keep controlling local hardware outputs. This makes the system more resistant to networking issues and server failures, giving administrators time to fix the problem without losing any information. Also, to a user, the system seems to be working normally and is still mostly usable.

Video monitor pipeline

The video monitor pipeline contains several different tasks with different resource needs and contains the main vehicle detection tasks.

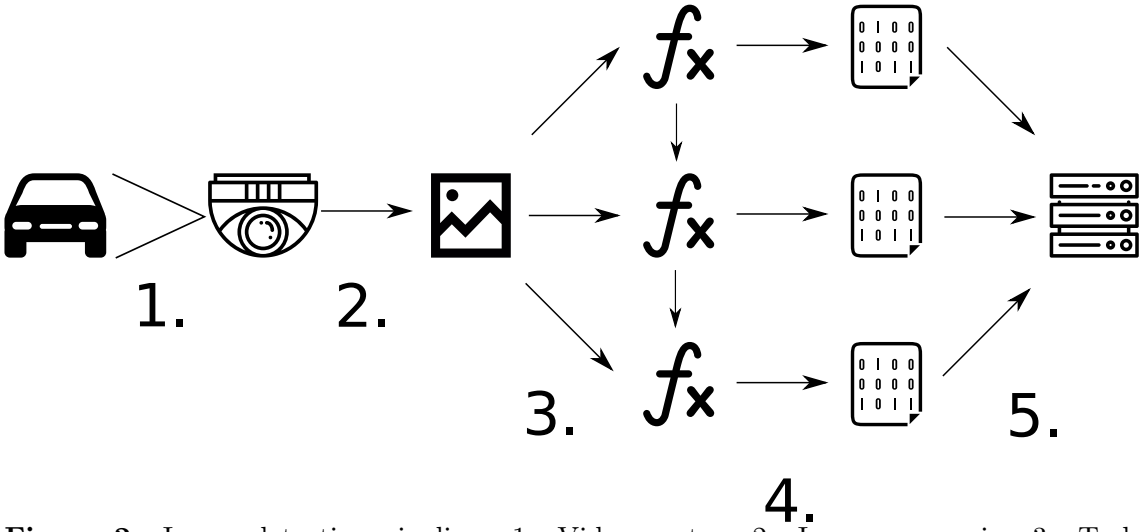

Figure 2: Image detection pipeline. 1. Video capture 2. Image conversion 3. Task startup 4. Image processing 5. Result sending

Figure 2 depicts the current video monitor pipeline from video capture to sending results to the server. The whole pipeline is run locally on the board.

The video pipeline is capable of handling 2 frames per second due to the cost of the image format conversion. More than that and the image conversion takes too much computing time, resulting in sub-par performance and even out-of-memory problems due to too many buffers in memory at the same time. This means that the pipeline drops every other frame when received to keep the frame rate in a reasonable range.

Step 1 is done in a separate program to separate the GStreamer dependency the video driver input requires from the main program. This makes it easy to isolate problems caused by possible memory corruptions or bugs in the video driver or GStreamer from the main program, enabling us to restart the video pipeline program without affecting the detection software. The GStreamer input module generates images in BGR565 color format.

Received frames from the video pipeline are sent over to the main program using shared memory and UNIX sockets. The main program then performs a color conversion from BGR565 to RGB using OpenCV's color conversion library [29] in step 2. This is done because all image manipulation and detection require the image to be in RGB-format.

Step 3 involves feeding the images to the task startup algorithm. The algorithm

contains multiple "monitors", which all get the image in turn. Each monitor can start a task for the image depending on what the image contains, e.g. interesting objects, movement or other anomalies, or may also start tasks after a certain amount of time has passed from the last task, e.g. snapshots for a timeline or forced state updates after a set interval.

Each started task is handled in step 4. Tasks may be run immediately or sent to a separate thread for processing. Tasks run immediately are very fast and usually take around a few milliseconds to process per frame. These mostly contain tasks running very light algorithms on the images or storage tasks storing the current frame to be further processed, e.g. encoded into a video file or sent to the server as is. Tasks sent to separate threads are either too slow to run for each frame, taking several seconds to run, or are dependent from past or future frames and cannot be run just on their own. Tasks may also start more tasks if the results need extra processing. These extra tasks are also processed in step 4, after all the others.

Step 5 sends results from the tasks from step 4 to the server. The tasks send the results to a communication service running in the main program which encodes and sends them to the server for further processing. This happens asynchronously and multiple tasks can send results at the same time.

2.2 Server

The server acts as a congregator for all the processed data the clients send. It further processes the data to create a timeline of interesting events in the monitored area and serves the current and past state for the user to see. One server must be able to support up to a few thousand clients at a time, which means that the data processing on the server must be very light for the system to not bottleneck performance on the server side.

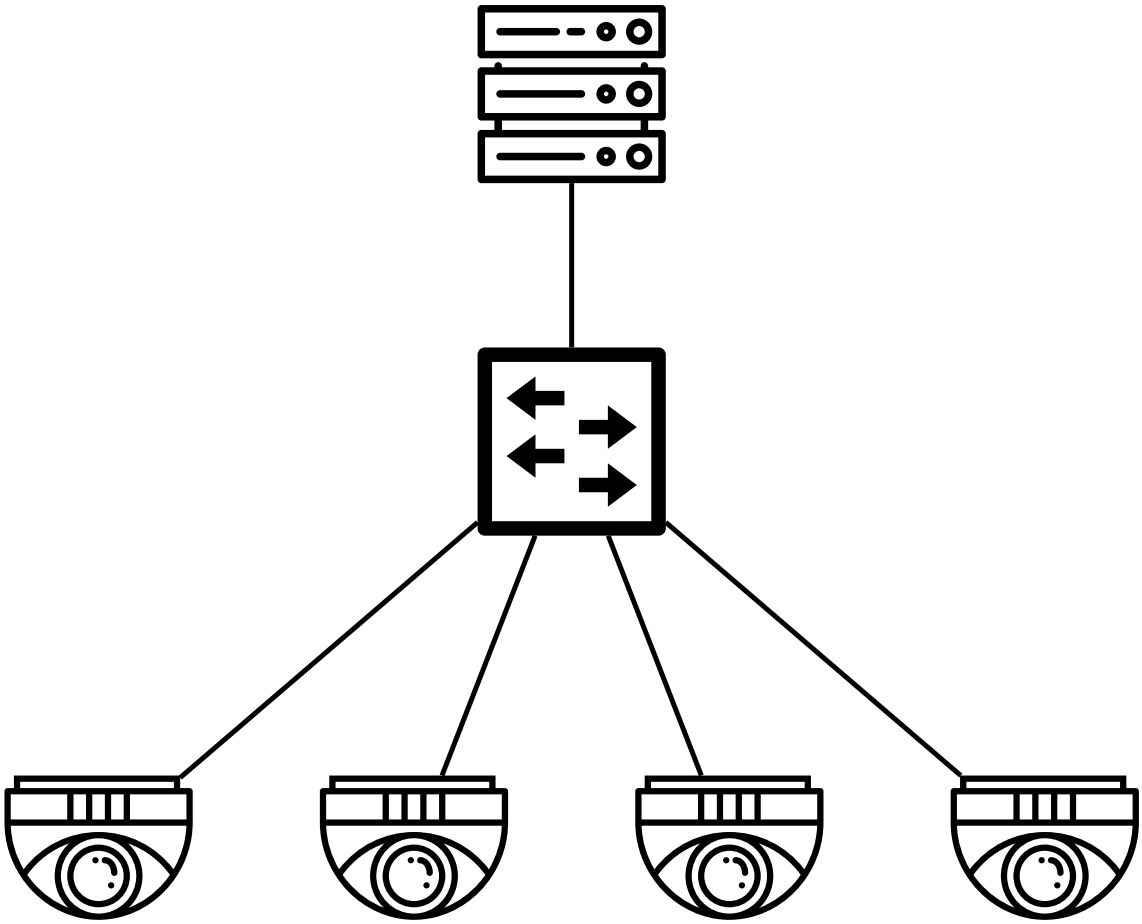

Figure 3: Sample network

Figure 3 depicts a sample network, with the server on the top connected to a network switch. Clients are connected to the switch and communicate with the server over it.

Currently, the server logs the movements of detected vehicles in the monitored area, combining detections as a timeline of events during its visit by matching the detected attributes of a vehicle with previous detection attributes. Videos and pictures are attached to the timeline for the user to search through using several filters based on the collected attributes. Several thousands of vehicles can be tracked at the same time in one system and can be combined even over different physical areas.

Several statistics can be further refined from the processed data, e.g. graphs for vehicle amounts over time in an area, average and mean times vehicles reside in one place and movement routes of vehicles. These are usually generated only on user request, but several hourly or daily runs are made ready for a quick overview of a running system.

The server also functions as a configurator and command node for all the boards. Boards get their configurations for monitorable areas and enabled sensors from the

server when they first connect. The boards can receive several commands from the server, including requests for individual boards to change their sensor configurations, send log files, request unusual sensor readouts, e.g. full resolution snapshot images from cameras or temperature readings, or command the client to either soft- or hard-reboot in case of a problem. The server also monitors which clients are active and notifies the administrators of possible outages in the client network.

All the server components are written in Golang [9], which while not the most performant language out there offers great abstraction between very low level protocols and high level server structure in our opinion.

2.3 Communication

Communication between the server and the client is done using gRPC framework [14] over TCP/IP. Data is encoded using Protocol Buffers [33] and utilizes both unary and streaming RPC-calls [15] to carry commands and data over the network.

The network in most cases is an isolated local area network (LAN), with 100 Mbps speed equipment. The server usually acts as a DHCP-server and a gateway for all the equipment in the network, but different configurations are supported.

The server also supports clients connecting through a Virtual Private Network (VPN) over the internet if a remote installation of one or several clients is required. This affect the available bandwidth with the unit, as the unit can connect from basically anywhere with internet access. However, these units are not usually configured to capture video, requiring only a very minimal amount of bandwidth to function.

In this thesis, we are mostly interested in the LAN installations, as these are the most common and demanding systems regarding performance. Also, measuring proper results for a remote client is very difficult as the measurements are heavily affected by the method by which the client is connected to the internet. A mobile connection over 4G/3G is vastly different to a physical LAN connection.

3 THEORY

3.1 Distributed computing

Distributed computing refers to computation that spreads over space and time. As an example, geographically separated database servers may each contain part of the distributed state of an application. [46] This grants a system a level of scalability not otherwise possible with centralized computation methods. The main characteristics of distributed systems are the lack of a global clock and the concurrency of each individual component such that each component can function and fail on their own. [44] A network of devices performing distributed computing is called a distributed system.

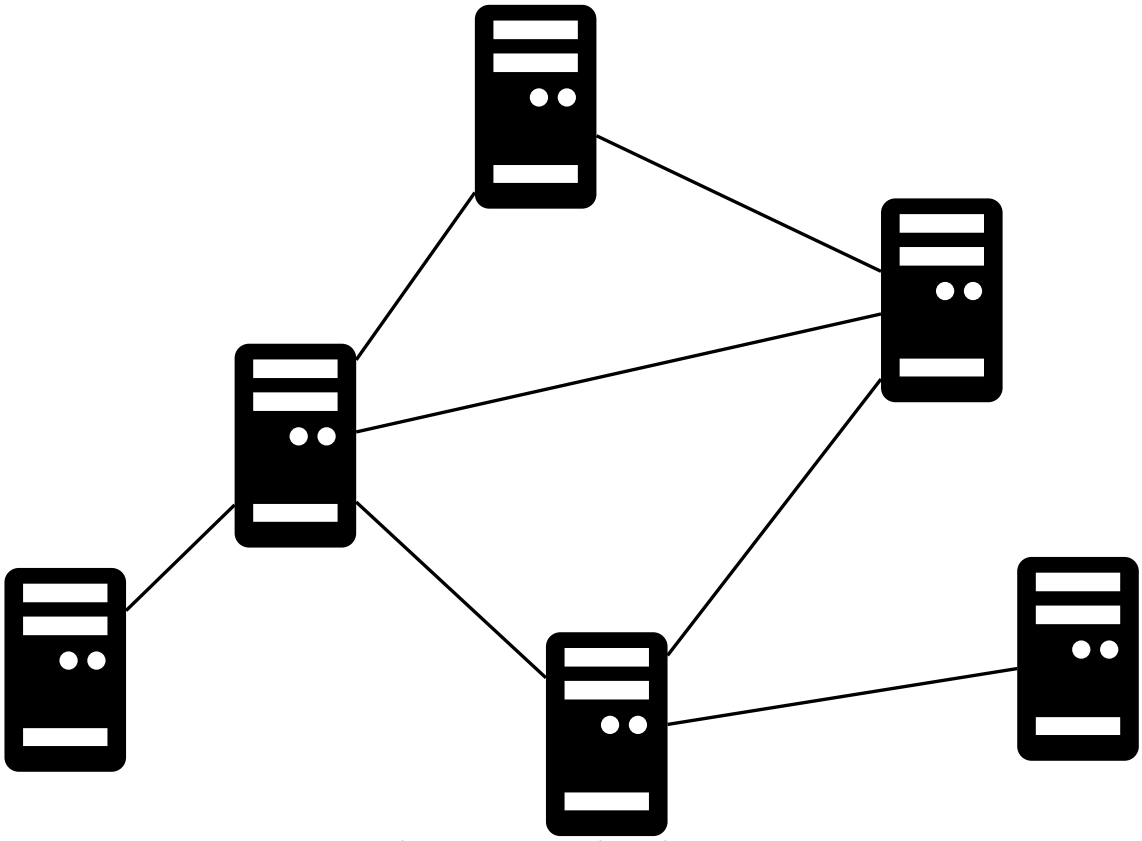

Figure 4: Distributed system

Figure 4 depicts a sample distributed system, a set of computing resources connected via network connection. Every component is connected to at least one other component, creating a connection graph.

Distributed systems consist of several independent computing resources, which are connected via some kind of network. [32, 38] These resources can vary in computing

power, storage and link speeds and may offer different services for the system. Different distributed systems may vary greatly from each other, containing just a few static computing resources, e.g. a small HTTP service with two instances for fault tolerance, to huge networks of dynamically available devices, e.g. cryptocurrency mining. The systems vary from having a clear centralized control to having a control hierarchy to having no control hierarchy at all [42].

Types of distributed computing

Distributed computing is a blanket term for any type of computing in which different parts of a program run simultaneously on different computing resources with each other connected over a network [38]. The term can further be narrowed as cluster computing, grid computing or cloud computing depending on the application and the physical connections between the computing resources.

Cluster computing is a term used when a group of computers are connected together to run as a single entity. [1, 37] This is the most tightly connected type of distributed computing, as a computing cluster is usually deployed in a single location connected with high-speed network equipment and is used to heavily parallelize tasks. Each unit in a cluster is usually identical to the next, offering constant performance on each. Clustering is an old term, dating back to the sixties [1]. A modern example of a computing cluster would be Oracle Solaris Cluster [31].

Grid computing is more loosely connected distributed computing than cluster computing, usually consisting of several different computing resources connected together over the internet from different locations. [37] This approach trades in performance for easier resource sharing. Units in a grid may vary in capabilities more freely than in a cluster, creating a more heterogeneous network of processing power. A good example would be the Folding@Home-project [11], which uses volunteering computing resources to simulate protein dynamics.

Cloud computing is the loosest form of distributed computing, offering on-demand computing resources with flexible resource provisioning and broad internet access [37]. Most of the computing resources in the cloud are virtual machines running in a server farm and share the same hardware with other virtual machines. Amazon EC2 [2] is a good example of this, providing easily tailorable computing resources for different use cases.

Differences to local computing

Local computing can be seen as the other kind of computing compared with distributed computing, with one computing resource working on its own tasks. The major differences between the two concern the areas of latency, memory access, partial failure and concurrency. [47]

Latency is the most noticeable difference between the two, with modern memory access latency being in the range of 13-24 nanoseconds [45] and average web server response time being 553ms with the best 10% achieving under 221ms in February 2019 [6]. This is 7 magnitudes of difference in response time. Designing around this increase in delay is a must for distributed systems as operations take significantly more time due to network delay.

Memory access itself is also completely different in local and distributed systems. Distributed systems different parts must communicate in some form over the connecting network to share memory and system state. A local system is not restricted to only using local storage, such as system memory or the filesystem, it doesn't need to synchronize its state over the network. A decision must be made when designing a distributed system at which level its memory access is shared. Fully sharing each parts local memory is not feasible, so a communication interface must be created. Also, depending on the use case, only part of the system may need a set of information at once, so broadcasting everything to everyone isn't always the right choice.

Partial failure is mainly an issue in distributed systems and not as much in local systems. As all systems are prone to some level of failure, be it hardware or software, error detection becomes much more complicated when considering distributed systems. As a distributed system consists of multiple hardware and software instances working together, the probability of an error grows exponentially with the number of participants in the system [36]. A local system can also detect and handle errors in a more straightforward way, as all the components are present in one location, but with a distributed system, the error handling must be constructed into the communication protocol itself. The error handling must handle cases that wouldn't normally be possible in a local environment, such as network outages taking only half of the systems resources down. In a local system, failures are either total or detectable by some central resource, but in a distributed system a total system failure is almost indistinguishable from a network outage [47].

Concurrency, while being a difficult task in both local and distributed systems, is built-in and required in a distributed system. While local systems use threading and

multiple processes to handle concurrency, it is again not required. Distributed systems by nature consist of multiple network resources, each with at least one process. Synchronization between concurrent tasks is also different, with local systems being able to use local mutual exclusion via shared memory and a common physical clock, and distributed systems needing special algorithms to reach mutual exclusion, such as Suzuki-Kasami's Broadcast Algorithm [43] or Ricart–Agrawala algorithm [10].

3.2 Load balancing

Load balancing is the act of distributing tasks to a number of workers. This can improve stability, performance and efficiency by using several computing resources in parallel with a reduced need for complex threading and better scalability. Load balancing is commonly used in popular internet services with multiple concurrent backend servers serving multiple customers at a time, bringing both performance benefits in the form of reduced loads per backend server and fault tolerance as in the event of a network or data center outage the system can continue to operate in some capacity as long as some of the servers stay online. This also enables smooth transitions and updates to server software as instances can be taken offline one at a time for updates and maintenance. [7, 12, 21, 34]

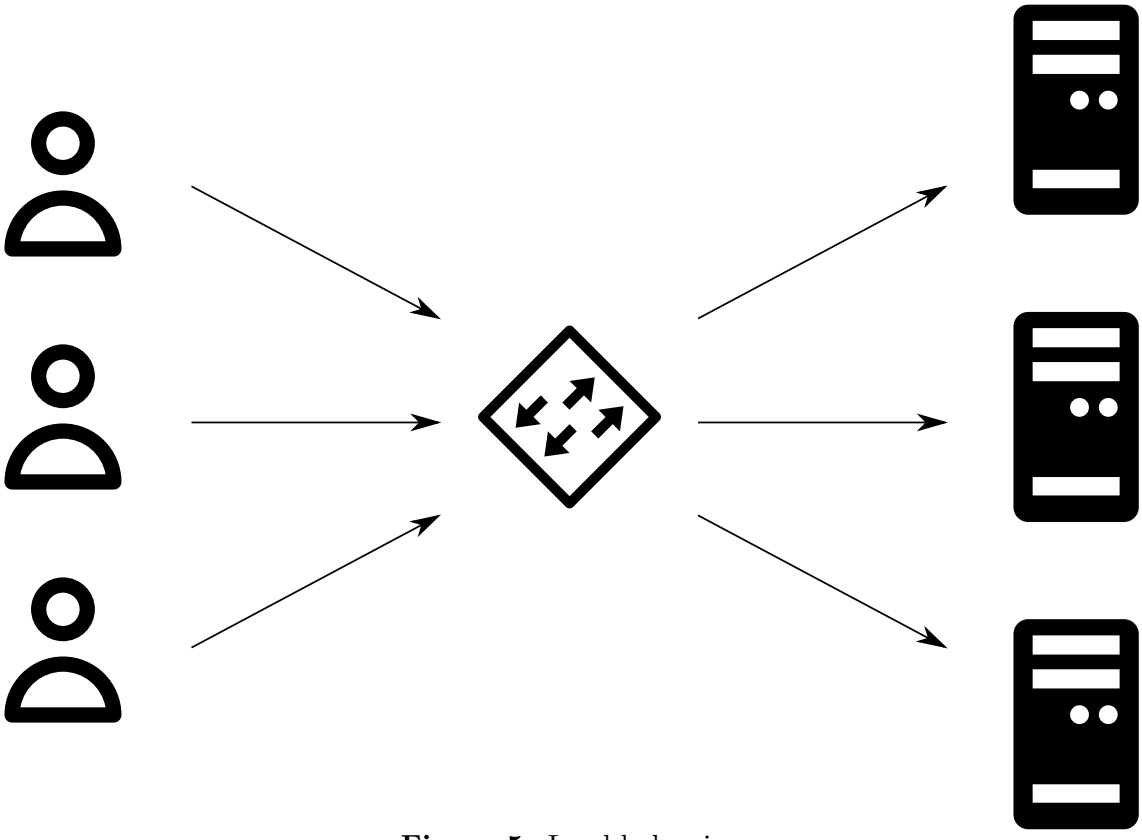

Figure 5: Load balancing

Figure 5 depicts a basic load balancing system. In the middle is the load balancer, which receives requests coming from clients on the left. The load balancer uses a load balancing method in order to decide a computing resource and sends the request on to the chosen server on the right.

Load balancing has several different balancing methods and many different metrics on which these methods are measured.

Load balancing methods

Load balancing methods or scheduling algorithms used by load balancers are divided into two different main classes: dynamic and static. [12, 25, 28] Dynamic algorithms use the current state of the system to decide which servers receive which tasks and can function differently depending on the situation. Static algorithms on the other hand don't use actual live state information about the system itself, using fixed rules and settings to schedule tasks to servers. In most cases, dynamic algorithms tend to be more complicated than static algorithms as the increased need for information about the systems state in dynamic algorithms requires data gathering methods of some kind in addition to the scheduling itself [25].

The attributes and state which dynamic algorithms take into account include available network bandwidth, current load, response times and performance of each server depending on which algorithm is used and if some attribute is being prioritized over the others. [12, 25] As an example, if response time is important, the scheduler may collect information about the response times and performance of each server, prioritizing servers with the shortest average or median response time.

Some dynamic algorithms may also be distributed as separate instances and configured to work as a group. [12, 28] This reduces the load on a single load balancer and prevents a single point of failure from forming. This however further increases complexity of the algorithm, requiring synchronization between each instance of the algorithm.

Examples of dynamic load balancing methods are ant colony optimization (ACO) [12, 27] and honey bee inspired load balancing (HBB-LB) [24].

Static algorithms use information configured beforehand to schedule tasks. This information may be configured by hand or may be generated by benchmarking each server separately before the load balancer is started. [28] Configurations may include manual priorities, capabilities of each server including processing power and memory, network connection statistics and geographical locations of the servers. While these

parameters can contain dynamic information, a static algorithm doesn't update its behaviour when the information is updated on the fly. [25, 28]

Examples of static load balancing algorithms are round robin (RR) [20] and opportunistic load balancing (OLB) [17].

Load balancing metrics

Load balancing algorithms can be measured in a myriad of different metrics. These metrics include throughput, response time, makespan, scalability, fault tolerance, migration time, degree of imbalance, performance, energy consumption and carbon emission. [12] Most of these focus on the pure performance the algorithm achieves, but some take environmental effects into consideration.

Throughput is the raw number of tasks an algorithm can handle per unit of time. This is most useful in a situation where the algorithm used tries to optimize the amount of tasks handled and/or there is a set number of tasks to be processed. Closely related is performance, which tells how much computing power the algorithm takes to schedule tasks.

Response time and makespan are two metrics measuring task computation time, response time measuring the average or minimum amount of time required to schedule a task and serve a response to the requester and makespan measuring the maximum time for the same case. These are most useful in cases where the system needs to be as responsive as possible.

Scalability measures the capability of the algorithm to work with a different number of servers. A highly scalable algorithm works well with any number of servers and can tolerate changes in server numbers over time.

Fault tolerance determines the capability of the algorithm to react to a failure in the system. Failures can include server or network outages, running out of resources or even errors in the algorithm itself. A highly fault tolerant algorithm is useful when uptime is a priority.

Migration time measures the time the algorithm takes to transfer tasks from an overloaded server to other, non-overloaded, servers. This can be useful if the tasks last long, e.g. streaming data over time, and/or the number of servers can change on the fly, creating new server instances as the old ones get overloaded.

Degree of imbalance tells how fair the load balancing algorithm is. A low degree of imbalance means that all the servers have an almost identical amount of load at all

times regardless of differences in task length and performance need.

Energy consumption and carbon emission are closely related, but used in different situations. Energy consumption measures the amount of energy the system takes, as more balanced and low performance needs require less power, resulting in less heat generated and longer battery life in mobile servers. Carbon emission takes into account the generated emissions per each server.

4 COMPUTATION UNIT

Computation unit is the new client software package we will be creating to handle offloaded tasks from any unit in the network. The computation unit will have a gRPC server for incoming task request, a gRPC client for communicating with the main server and the main application which runs received tasks.

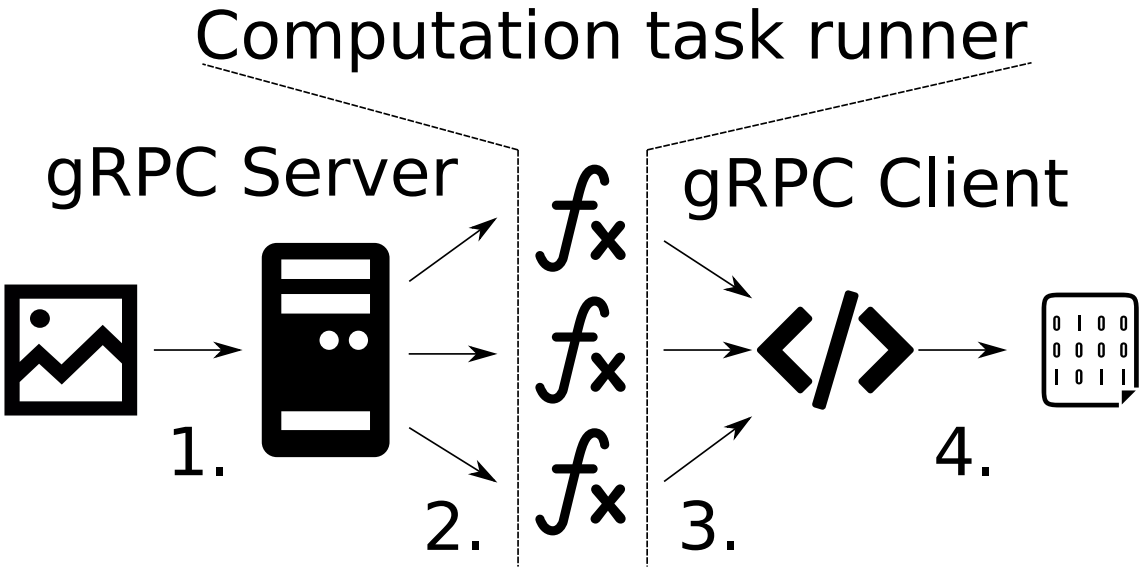

Figure 6: Computation unit. 1. Image processing request 2. Starting computation 3. Result generation 4. Response sending

Figure 6 depicts a computation unit with its processing steps marked with numbers. Each of these and the corresponding steps are described in this chapter.

4.1 gRPC Server

The server listens to any incoming detection request, runs the detection algorithm and returns the same attributes as the client would have generated on its own. The message will be passed as a simple unary RPC-call. Step 1 in Figure 6 corresponds to the reception of detection requests. When compared with the pipeline in Figure 2, the computation unit doesn't have a dedicated image source, instead relying on the client to provide all needed data for a runnable task.

The bottleneck for the unit in most cases will be the network itself. While under the original design all units would process their own data when needed and only sending data over the network when changes were detected. However, when using the computational client, only a very limited filtering is applied to the data before it is sent over to the computational client.

The data also suffers in image quality as sending raw image data over the network would be inefficient. The data is compressed using JPEG-compression [19] using OpenCVs imencode-function [30]. The effects of this on the accuracy of detections are very slight and are not considered further in this thesis.

The server also has a status endpoint from which the server occasionally fetches a ”heartbeat” message from. The message will contain stat from the units including a number of processed images, average detection times and most importantly the possible disconnect which could have happened between the unit and the server.

4.2 Computation task runner

The computation task runner software uses the same image detection and task libraries as the main detection pipeline software. The task runner has two main parameters: the number of parallel task workers and the maximum number of jobs the server can assign it at a time.

The maximum number of jobs is communicated by the client on registration to the main server and acts as the capacity for parallel jobs the server can assign to a single unit. When the units' capacity is reached, the server will not send any requests to the unit in question, unless no other unit is found and the unit is configured as a backup. The job capacity is not a hard limit, but too many jobs at a time can cause the detection request to time out.

The number of parallel task workers dictates how many parallel threads are launched on startup, each listening for an input task. When a task is received, a random thread is assigned to compute the requested task. This is depicted in Figure 6 as step 2. Each thread can handle all types of tasks the runner supports. On task completion, the result is passed back to the server, which is depicted as step 3. The server then returns the result as a return value for the unary gRPC-call, depicted as step 4.

4.3 gRPC Client

The client side of the computation unit registers itself to the main server as a computation resource, sending the computation servers listening address and the amount of parallel computations it is prepared to do. This enables different clients having different hardware configurations to send an approximation of how much load it can handle. For the boards, this will most likely be one, as previously stated,

the client can only run one instance of the algorithm at a time. For the offloading server, we will be testing a few different values for this.

The client also has a separate call for signalling the server of a completed detection job. This is called after every detection to decrement the server-side counter of current parallel jobs running on the unit.

4.4 Hardware

The offloading client, while not being able to gather sensor data on its own can be run on any hardware platform with an internet connection. The software will be deployed on a separate server platform with a dedicated GPU, which brings several times the computational power a network of hundreds of units. Being able to use any hardware, including any number GPU:s with CUDA capabilities, makes deploying the client very flexible and adaptable to almost any computing load.

The following table contains the hardware used in this thesis for the offloading server:

CPU Model	Intel(R) Xeon(R) CPU E5-2620 v4
CPU Cores	8 (16 virtual)
CPU Clock speed	2.10 GHz
GPU Model	Nvidia GeForce GTX 1060
GPU CUDA cores	1280
GPU memory	6 GB GDDR5
RAM	16 GiB

Table 2: Offloading server hardware

Table 2 contains information on a rather basic server configuration with an additional GPU to boost the performance of image processing algorithms. The server runs an installation of Arch Linux [3] with CUDA enabled version of OpenCV [5] version 4.2.0. This enables us to utilize the GPU automatically when compiling the computation unit software using this version of OpenCV.

The server will be connected to the testing network using Gigabit Ethernet connection through a Gigabit Ethernet network switch.

5 CHANGES TO THE CURRENT SOFTWARE

This chapter describes the needed changes for the data collection client software.

5.1 Algorithm partitioning

The main programs' video monitor pipeline must be partitioned such that the functionality of the board does not depend too much on the new computation units, as the main programs offline functionality is on a very high priority. We will be using Figure 2 as the basis for the partitioning.

Step 1 will remain on the board as sending each frame over the network would be too excessive and would probably overload the network fairly quickly as one raw frame takes $\sim$ 10.1 megabytes of memory. This would generate $\sim$ 20 megabytes per second of traffic per board, taking only one board to overload a 100Mbps network switch, which can handle approximately 12.5 megabytes of traffic per second.

Step 2 will also remain on the board, as the converted images take even more space, requiring $\sim$ 15.1 megabytes of memory each. At this stage it would be possible to encode the whole image as JPEG, but while this drops the space required to $\sim$ 1.4 megabytes, it would still take only $12.5/1.4 =\sim 9$ units to overload the switch.

Step 3 and 4 contain the task startup, where in some cases we are only interested in small parts of the previous steps input image. Most of the tasks are so fast and require such little amount of processing time that it would not be in any way worth it to send them over the network to be processed. However, there are some tasks, mostly the tasks moved to the background threads due to their lengthy processing times, that the most logical place to do the partitioning is here. Step 5 is too fast and requires little processing time, so it is not worth sending it to the computation units.

We identified two tasks in steps 3 and 4 clearly taking the most amount of time in the detection pipeline, which we will call Task A and Task B. The tasks use different image inputs and generate different outputs containing information about the parked vehicles in the image. Task A is run on each received frame and Task B can either be triggered by Task A, or after a set amount of time has passed since it was last run on an area. Thus, multiple instances of both tasks may be started on one frame. The tasks can handle only one vehicle at a time, so if multiple vehicles are found in a single image, multiples of both are queued for the background thread to process. This is basically a busy loop where each vehicle is iterated over one

at a time, looping after the last vehicle has been processed. There is little to no advantage to detect if the image has been changed or not between the runs of the same vehicle as it would bring unnecessary complexity to the pipeline. The little response time improvement would be lost as soon as we take on account that if a part of the image has been changed where a vehicle is located, usually the whole image has changed as most passing vehicles cause severe glare in the image with their headlights.

Both tasks only use a very limited part of the input image, dropping the encoded data size to 100-500 kilobytes. Both tasks are also heavily parallelized and benefit from extra processing cores and CUDA capabilities a GPU provides.

We will be adding a simple movement detection algorithm before the steps to filter out unchanged images from the pipeline. The algorithm detects if anything in the frame has moved compared with the previous frame by doing a simple matrix subtraction between the two frames and only sending the changed frames to be recognized. This has not been done before as it would have brought unnecessary complexity and no real performance benefits as the whole image is usually reported as moving when a change happens. The movement detection algorithm is already being used in other parts of the image processing pipeline for other event monitors, so the movement information is already calculated as the image arrives at the Task A or Task B startup monitor. This also frees up the computational resources used normally by these tasks for the computation unit software.

5.2 Client pipeline changes

The client side algorithm for Task A and Task B will completely be removed and moved to the computation unit software. An instance of the computation unit software will be run as a separate process for the Task A and Task B runners. The monitors for Task A and Task B will contain a gRPC-client, which can connect to both the main server for computation unit requests and the received computation unit for the request.

In case of a failure to obtain a free computation unit from the server, being it from the exhaustion of a free computation unit or a communication problem, the client software contacts the local computation unit software through a loopback interface. This should ensure that the unit behaves as before the change with the important offline functionality in mind.

5.3 Load balancing methods

The load balancing method, or scheduling algorithm, in this case is by design dynamic having to take the systems state into account such that no computation unit receives tasks in excess to its parallel limit. The system is basically a master-worker configuration, as all the computation units report and receive all their tasks through it.

The algorithm must also be compatible with the fact that if no free computation unit is available when one is requested, the scheduler must be able to tell that this is the case and communicate this to the requesting party, telling it to use its configured fallback computation unit instead, which is usually the unit itself.

We came up with six different algorithms to benchmark, which are all described in the following sections. The algorithms, as they are in this thesis, are the first few basic scheduling algorithms we could come up with ourselves. In the future, a proper scheduling algorithm may also be used, but will need to be customized to support the extra functionality as described before. A random scheduler is also benchmarked, giving a decent reference time for the schedulers to beat. The random scheduler cannot be used in the application as we could not satisfy the 'no free computation unit' constraint in any meaningful way using it.

In the following sections, a green square represents a computation unit with free capacity and a red square a computation unit with no free capacity. An arrow running from the scheduler represents the order in which the scheduler considers the computation units and the symbol S represents the scheduler. If a figure contains other symbols or marking, they are explained in the relevant section.

General functionality

The general functions a scheduling algorithm in the system must implement are the insertion and deletion of computation units, scheduling a free computation unit, freeing a scheduled computation unit and being able to tell when there are no free computation units to schedule. All of the implemented schedulers store all their computation units in a single list, a slice [13] in Golang, and perform operations using this list. This is done to make benchmarking and implementing the different schedulers easy, but more advanced schedulers could in theory use other data structures to store their computation units.

The insertion and deletion of computation units is done the same way in each scheduler.

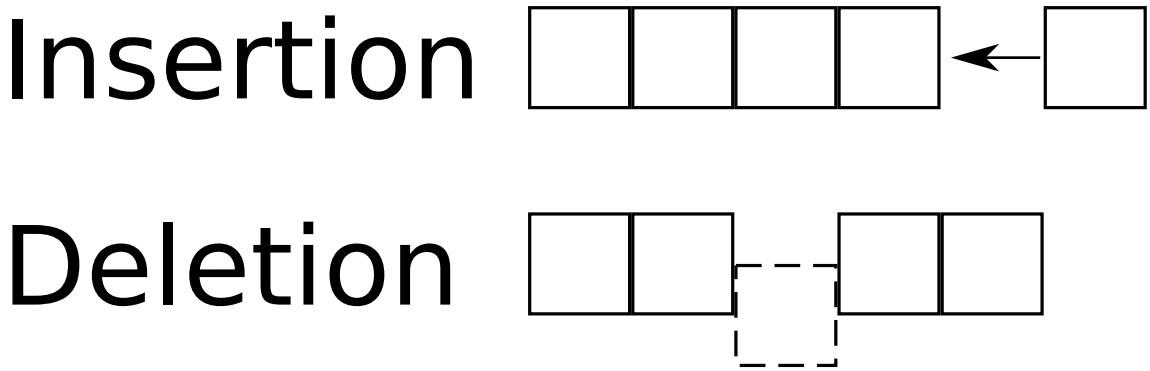

Figure 7: Insertion and deletion

Figure 7 depicts the process of inserting and deleting units from the list. A new unit is appended to the end of the list and in the case of an empty list, assigned as the first element of the list. Deletable units are deleted in place and garbage collected by the Golang runtime [35].

Each scheduler must test each computation unit in the list only once per scheduling for free computation capacity. If all the units are fully reserved, the scheduler must schedule an invalid computation unit, signalling the requester to use a fallback computation unit. This is not part of the performance benchmark, but must be implemented by the scheduler.

Freeing a computation unit is done the same way by all the schedulers and is not part of this benchmark.

Random scheduler

This reference scheduler is used to give a reference time for the rest of the schedulers.

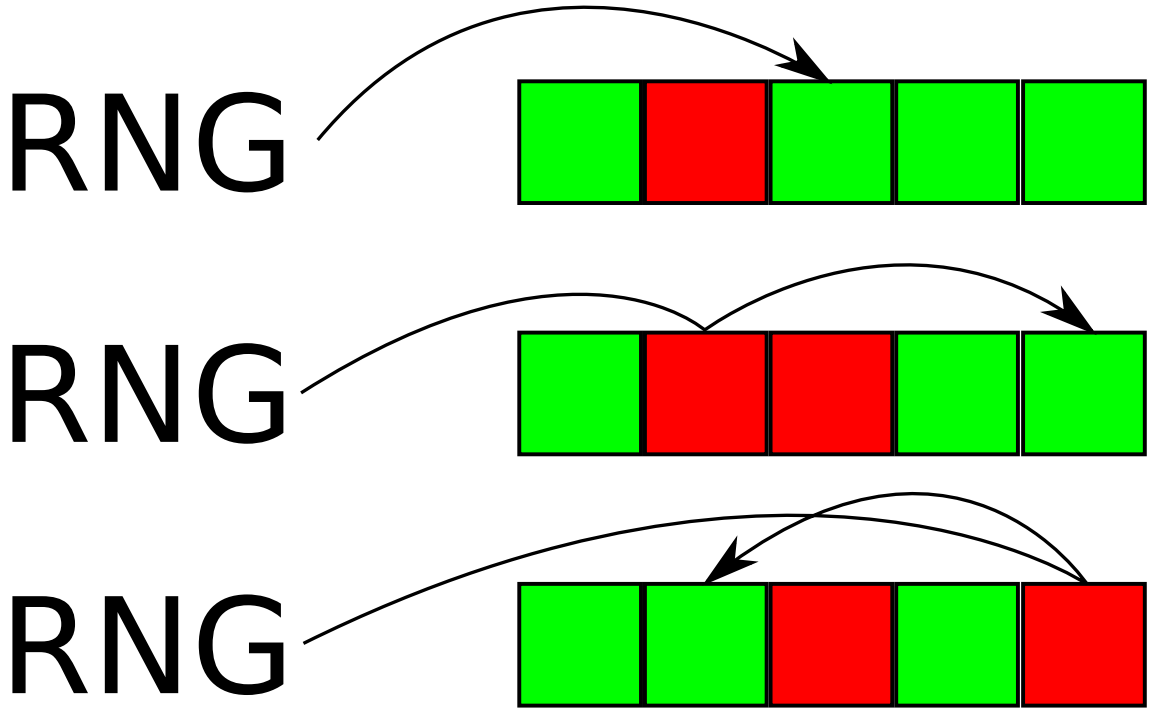

Figure 8: Random scheduler

Figure 8 depicts the random-scheduler. The *RNG* in the picture is a random number generator which generates a random index from the computation unit list. The scheduler uses the generator to randomly pick units from the list until a unit with free capacity is encountered. This unit is then reserved and returned for the requester.

This is only a reference scheduler as it does not satisfy the 'check each unit in the list only once' requirement and it cannot tell when the list has no free units left either.

First-free scheduler

This is the most basic scheduler we could come up with.

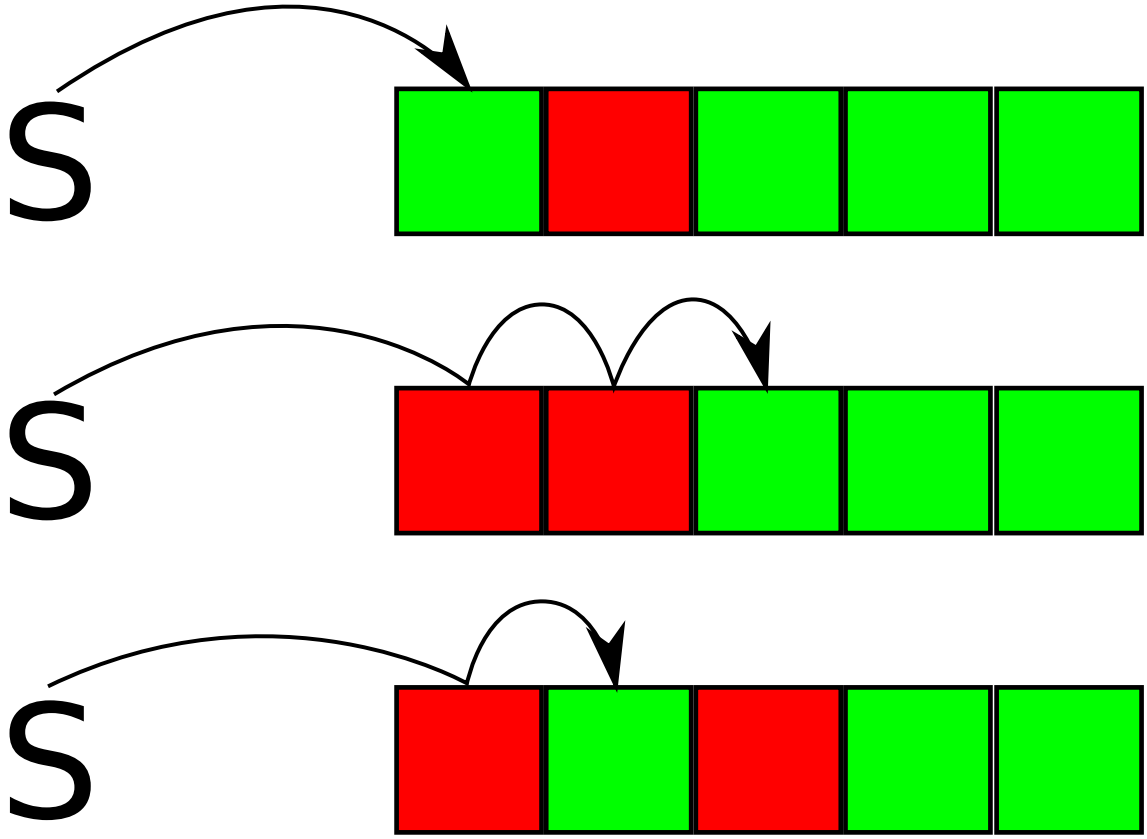

Figure 9: First-free scheduler

Figure 9 depicts the first-free scheduler in action. The list is scanned from the beginning to the end, returning the first free computation unit it finds. If the end of the list is reached, an invalid computation unit is returned. Having no internal state and no operations other than iterating on the list, it is the most basic of the schedulers.

Next-free scheduler

This is a more advanced version of the first-free scheduler, introducing a state where the last scheduling left off. This is similar to a round-robin -scheduler [4].

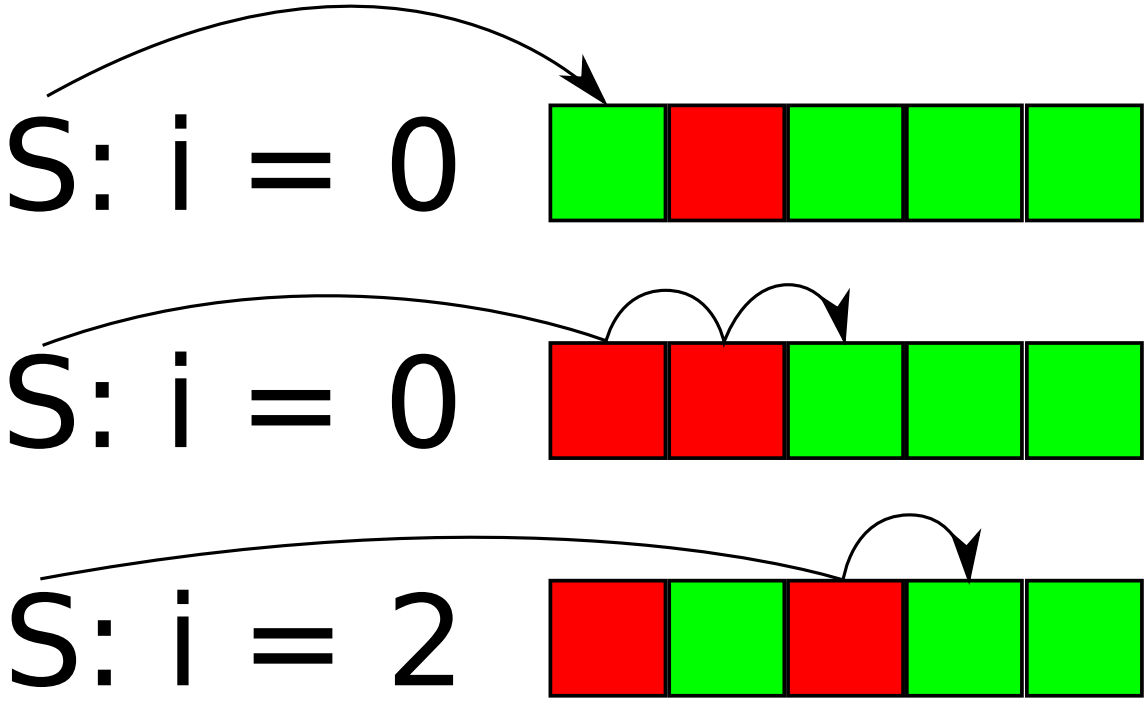

Figure 10: Next-free scheduler

Figure 10 depicts the next-free-scheduler and its internal state. The additional $i = n$ next to the scheduler depicts the last index the scheduler was in the last scheduling attempt. Every time the scheduler scans the list of computation units it starts from this index. If reaching the end of the list, the scheduler starts from the beginning of the list and scans until the same index is reached. If no free computation unit is found, an invalid computation unit is returned.

The index is incremented every time the scheduler encounters a fully reserved computation unit. This means that the index is left on the last reserved computation unit after one has been found. This is done to allow the scheduler to fill each computation unit to the maximum number of jobs the unit can handle before scheduling to the next unit.

Shuffle-first-free scheduler

This scheduler is almost the same as the first-free scheduler with the addition of shuffling the list.

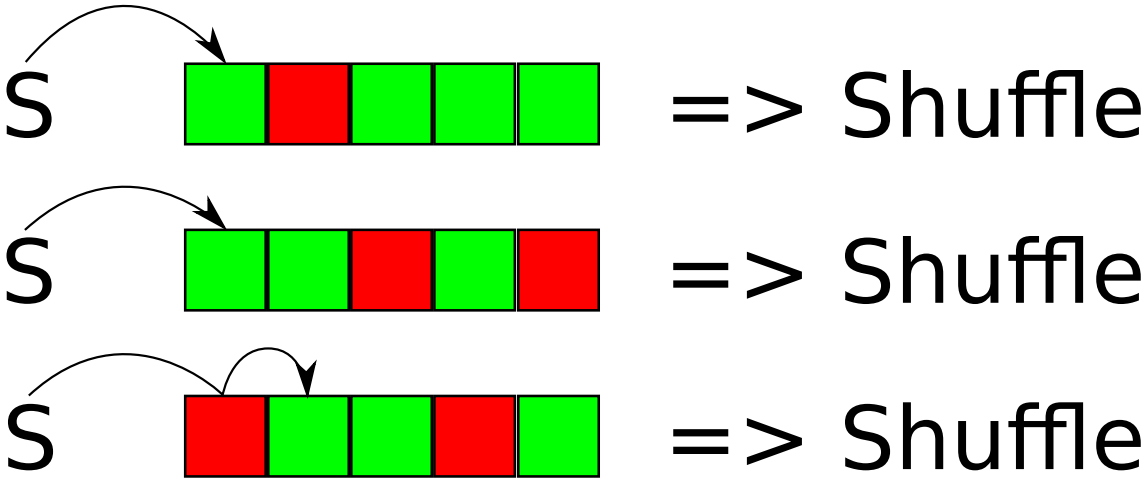

Figure 11: Shuffle-first-free scheduler

Figure 11 depicts the shuffle-first-free scheduler. The scheduler shuffles the list of computation units each time it has either found a free computation unit or has reached the end of the list. If the end of the list is reached, an invalid computation unit is returned.

This approach uses the same strategy as the random scheduler, but only checks each unit once. Randomly selecting a free computation unit is in theory faster than always iterating the list in the same order from the beginning. One of the pitfalls of the first-free scheduler is that the first few units of the list are the most likely to be fully reserved. This scheduler avoids this problem by shuffling the list each time.

The act of shuffling the whole list is on the other hand a rather heavy operation and may cost us more than it is worth.

Shuffle-on-completion scheduler

This scheduler tries to minimize the amount of shuffles it must perform while still benefiting from it.

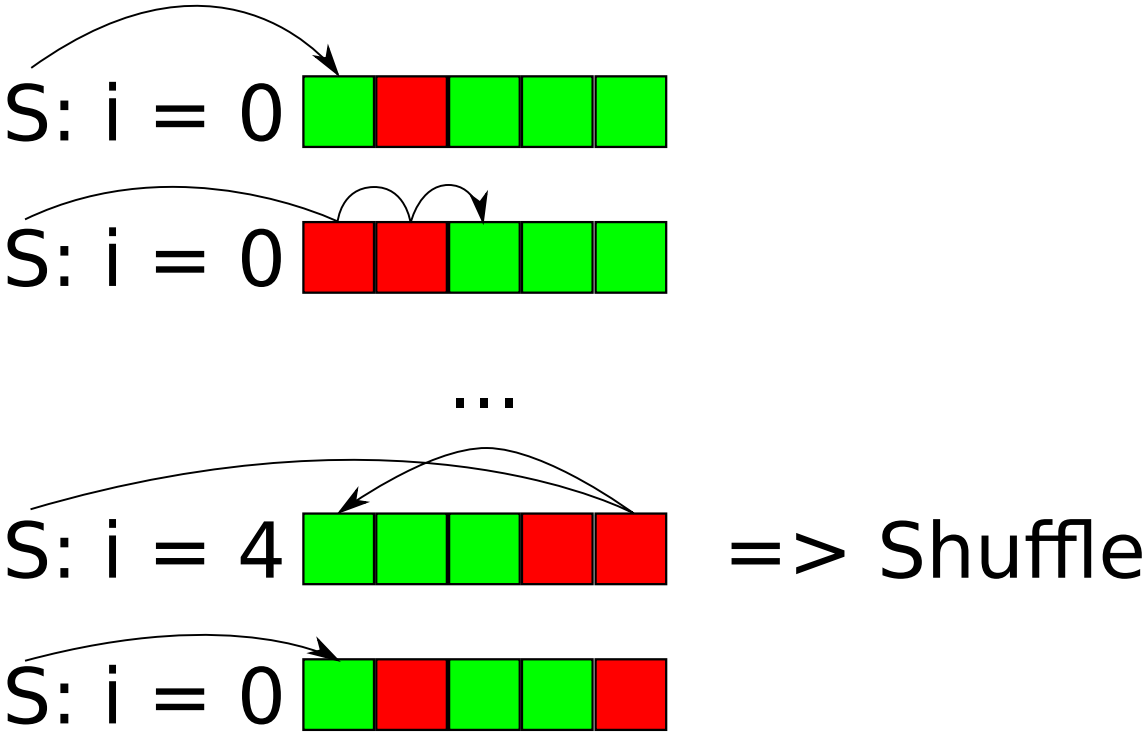

Figure 12: Shuffle-on-completion scheduler

Figure 12 depicts the shuffle-on-completion scheduler. The additional $i = n$ next to the scheduler depicts the last index the scheduler was in the last scheduling attempt. As in the next-free scheduler, the scheduling starts from the index and loops back from the start of the list if the scheduler reaches it. In addition to the functionality of the next-free scheduler, the scheduler shuffles the list when it reaches the end of the list.

This approach should have both good sides from the shuffle-first-free and next-free schedulers and should avoid the bad side of shuffling the list too often the shuffle-first-free scheduler has.

This shuffling of the list should also make our scheduler schedule jobs more uniformly over the network as it prevents fast and slow units from clumping up together in the list. This clumping is a real possibility in a system, as slow units register themselves slower than faster units, creating areas of slow and fast computation, which is not preferred.

Sort scheduler

This scheduler uses the performance of different computation units to sort them, trying to maximize the response time for each task.

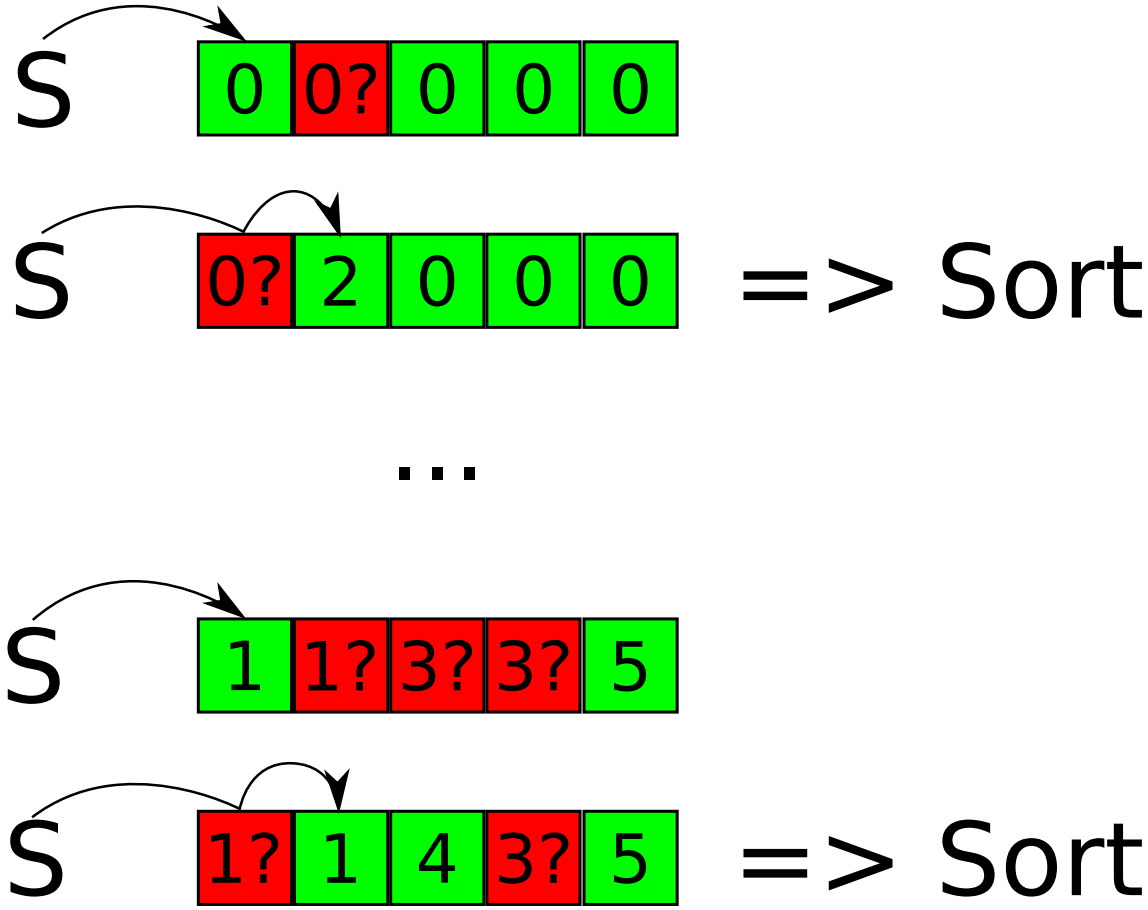

Figure 13: Sort scheduler

Figure 13 depicts the sort scheduler. The numbers on top of the computation units are the average computation times of each unit respectively. A time with a question mark represents that the unit is computing and the computation time will be updated as soon as the unit has completed its computation. While a unit is in this intermediate state the last value it had, represented by the number next to the question mark, is used as its average computation time.

Each computation unit start with an average computation time of 0. Each time the scheduler schedules a computation unit from the list it also checks if the list is sorted. If the list is not sorted, the scheduler sorts it. The sorting is made so that the unit with the lowest average computation time is first in the list.

The scheduler always starts at the start of the list, iterating from the fastest unit to the slowest. If no free unit is found, the scheduler returns an invalid computation

unit.

The sorting of the list is rather heavy operation and may cause this scheduler to perform poorly, but it improves response time for the tasks it schedules the computation units for, giving the fastest computation units first. This is not measured as part of the benchmark, but if the performance is not too much worse than the other schedulers, this scheduler could be a good candidate.

This also answers the clumping problem, but instead of spreading the clumps randomly, it sorts them in a preferable order, taking advantage of the different computation times different units have.

The sorting is made by using Golangs sort-packages [39] sort.Slice, which seems to use a combination of quicksort and insertion sort. [40, 41]

Sort-partial scheduler

This scheduler tries to optimize the amount of sorting it must perform compared with the sort scheduler.

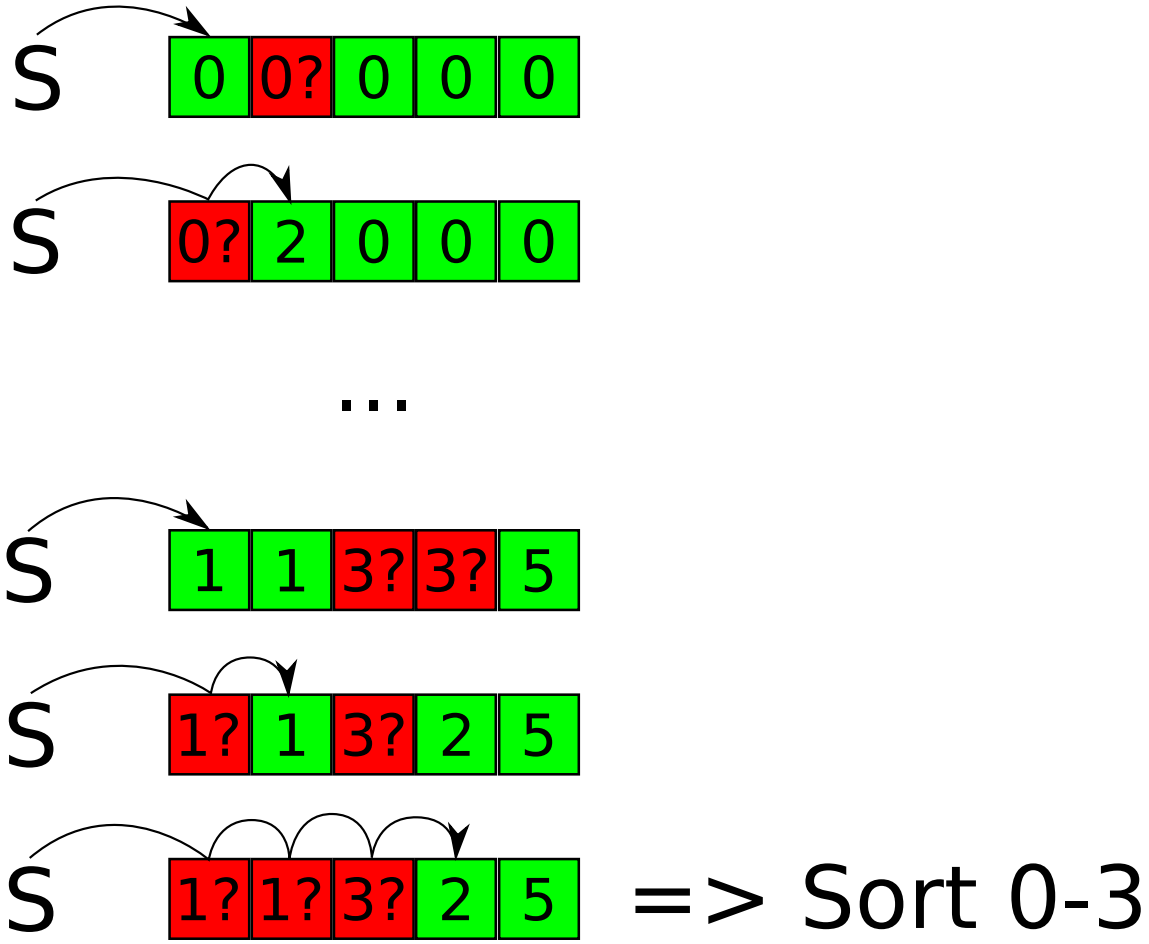

Figure 14: Sort-partial scheduler

Figure 14 depicts the sort-partial scheduler. As in the sort scheduler, the numbers on top of the computation units are the average computation times of each unit respectively. Question marks represent times that will be updated soon and while this state is on, the number next to the question mark is considered the average computation time of the unit.

Each unit starts with an average computation time of 0. Each time the scheduler schedules a computation unit from the list it also checks if the list is sorted from the beginning of the list to the currently schedulable unit. If it is not, the scheduler sorts the checked sub-list. This means that the scheduler only has to sort a small subsection of the list at a time, improving performance.

This sorting of only a subset of the whole list means that the scheduler almost never checks if the units at the end of the list. Usually, these are the slowest units, but in the case of an insertion or if the first few units handle the whole systems load, the last units in the list may never be scheduled to. For example, if the fastest units happen to be at the end of the list, the average response time of the system may suffer greatly.

6 BENCHMARKS

The following benchmarks include the current performance of both Task A and Task B, two different stress tests and benchmarks for the algorithms described in Chapter 6.

The current performance of the tasks is recorded from a production device running the tasks over a few minutes and the benchmarks are run on two different processor units and the offloading server. All of these connect to a single camera unit server as computation units. A separate machine will be used to run benchmarking software, which requests computation units from the camera unit server and sends a computation job to the reserved unit.

The two benchmarks run were a sequential one-job benchmark on a random unit to calculate the average computation time on each unit, and a parallel stress test to find the maximum number of jobs the unit can handle at once.

The algorithm benchmark runs several simulated workloads on all of the algorithms. An additional semi-realistic benchmark was also run using the results gathered from the task benchmark results to simulate a real system with realistic computation times.

6.1 Current performance

The current performance of the algorithm was recorded over a few minutes on a production board. A custom version of the software was pushed to the device for it to be able to record computation times for both Task A and Task B. The results are as follows:

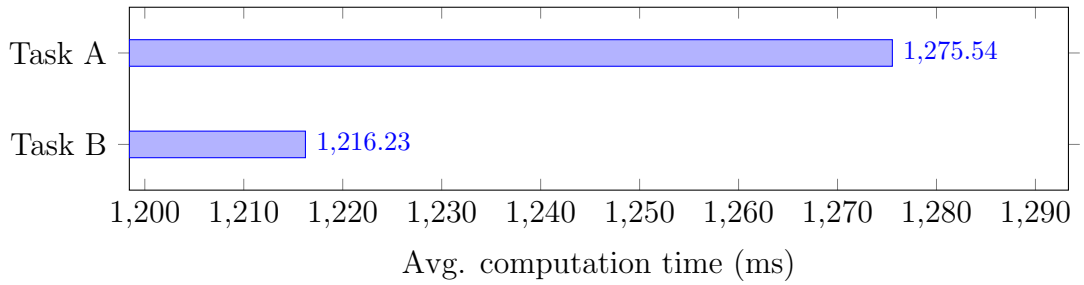

Figure 15: Current performance

As can be seen from Figure 15, both Task A and Task B take a considerable amount of time to compute. As multiple instances of both can be triggered on one frame and the frame rate being 2 frames per second, we can clearly see that one board is

not able to compute all of the possible spawned tasks.

6.2 Sequential average test

The first benchmark is the sequential, single job run, sending different jobs one at a time to random units registered on the camera unit server as a computation unit. The test measured an exponential moving average [23] using the following equation:

$$S_t = \begin{cases} Y_1, & t = 1 \\ \alpha * Y_t + (1 - \alpha) * S_{t-1}, & t > 1 \end{cases} \tag{1}$$

Using the equation (1) we can calculate the average only using a single floating-point variable to store the average, resulting in minimal extra memory usage. The α used in this benchmark was 0.1.

The following moving averages were recorded:

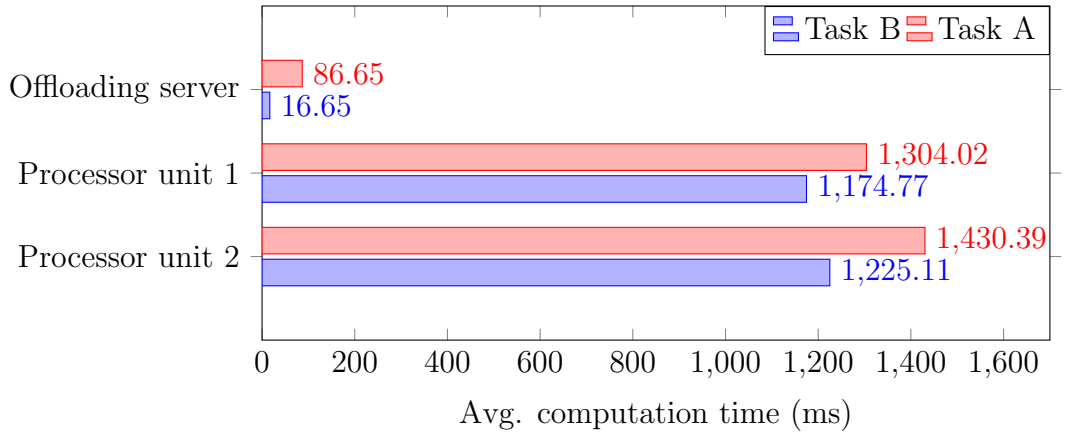

Figure 16: Single job performance

We can see from the results in Figure 16 that the tasks are on estimate $\sim$ 15.7 times faster with Task A and $\sim$ 72.1 times faster with Task B. Using these results, we can estimate that a single offloading unit can replace 15 boards in a system. We can also see that not all boards are equal, having non-trivial differences in their performance. This drives the case for the sorting scheduling algorithms, as faster boards would be scheduled more often, increasing response time.

6.3 Parallel stress test

The second benchmark is the parallel stress test, run only for the offloading server. This test runs jobs as fast as possible on the unit with different amounts of parallel worker threads to measure the maximum load the unit can handle.

As the unit has 8 physical CPU cores and 16 virtual CPU cores, the test was run with 8 and 16 threads with the following results:

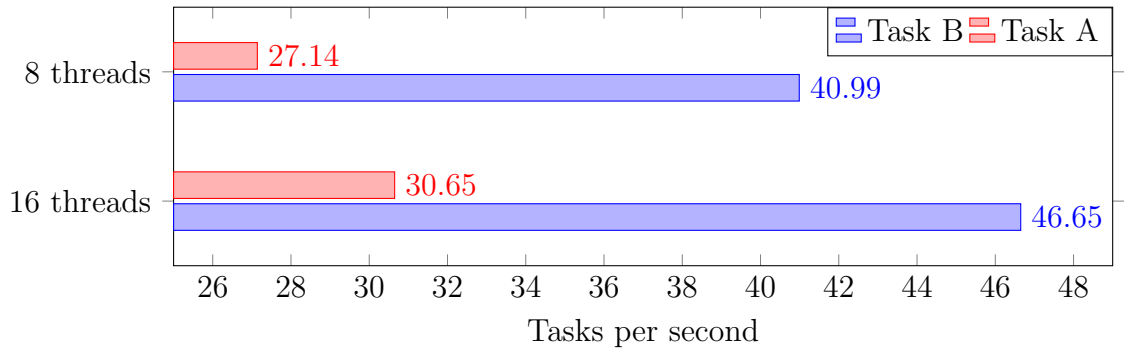

Figure 17: Multiple job stress test

From Figure 17, we can see that the estimates made in Section 6.2 do not reflect the actual number of jobs per second the offloading server can run. This is mostly due to the better utilization of all the processing cores the offloading unit has and the performance is further increased with the virtualization of the cores. From this test, we can further estimate that the real number of boards the offloading server can replace is somewhere between 15 and 30.

6.4 Scheduling algorithms

This section describes the benchmarks the scheduling algorithms were run through. The first subsection describes the test bench and the subsequent subsections the results collected.

Test bench

The test bench consisted of several runs of each algorithm against a list of dummy 'workers'. Each worker had three parameters: job capacity, task duration and task duration standard deviation. Task duration and task standard deviation were hard-coded as ten milliseconds for the duration and five milliseconds for the standard deviation. Job capacity was changed in each run ranging over 10, 5, 3, 2 and 1

capacity each. The task duration was further modified by applying the standard deviation to it, simulating workers of different speeds.

Each worker had a reserve method, which would return either true or false depending on if it had remaining job capacity left. If it had, a timestamp was recorded after which the capacity would be freed.

Test runs were made to a different number of workers, ranging over 10000, 1000, 100 and 10 respectively. This resulted in $4 * 5 = 20$ test runs per scheduler. The test recorded the total time the scheduler took to schedule 100000 jobs to the workers.

An additional 'Semi-realistic' run was also made using numbers gathered from benchmarking the current system. The test consisted of 500 workers as 'boards' with 1300 milliseconds as task duration, 100 milliseconds as the standard deviation and job capacity of 1. Additionally, one worker was simulating an offloading server with 40 milliseconds as task duration, 10 milliseconds as the standard deviation and job capacity of 8.

Test run 1: 10000 workers

The first test run was made using 10000 workers, which is an amount double to the required units the world's largest car park [22] at the moment of measuring. This test can schedule all 100000 jobs when at the job capacity of 10, so it should show the purest performance difference between each algorithm of all the tests as the workers themselves do not bottleneck the processing.

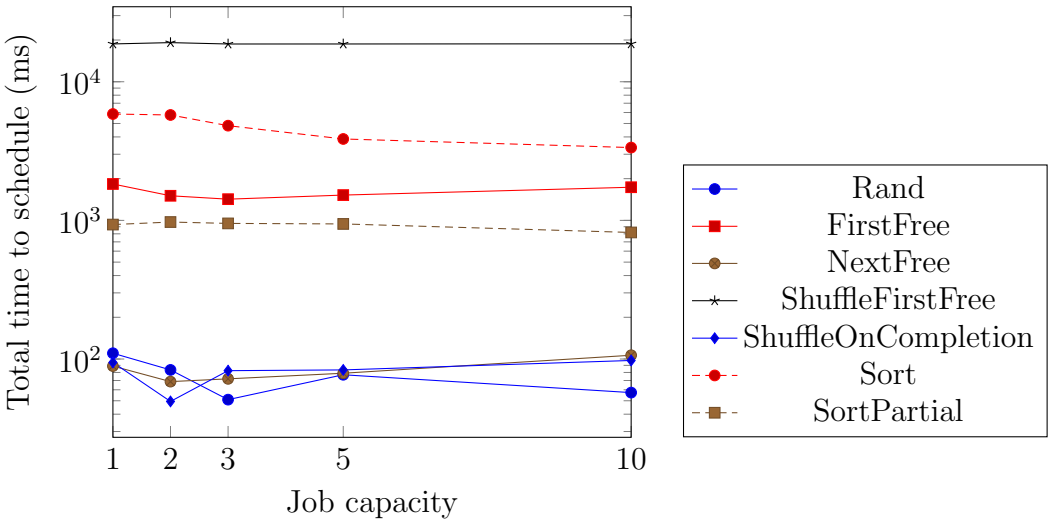

Figure 18: Algorithm test run 1: 10000 workers

From Figure 18, we can see that the algorithms group up a bit between similar

algorithms. Both NextFree and ShuffleOnCompletion have the same performance as both have the index where they left off last time stored, minimizing the number of unnecessary checks both indexed algorithms must perform per worker. Both of these also seem to perform as well as the pure random algorithm, which should be the fastest one as workers are readily available so selecting any worker randomly from the list should return a worker with free capacity.

The next three algorithms with similar functionality are FirstFree, Sort and SortPartial, with Sort hovering about 2-5 times slower than the other two. This is mainly because the amount of sortable workers is so large the sorting algorithm takes a while per sort. SortPartial is about 2 times as fast as FirstFree, which is expected as the reduced amount of sortable workers per sort and the advantage the algorithm gets by ordering the fastest units first does add up over multiple schedulings.

ShuffleFirstFree is clearly the slowest of the algorithms here, mostly giving a steady performance not depending on the job capacity of the workers at all. This may be because of the sheer amount of randomizable workers in this benchmark.

Test run 2: 1000 workers

This run with its 1000 workers is close to the amount of boards the currently largest existing installation of the system has. This is a good benchmark for those systems as they tend to be the most demanding, requiring a high-end server to perform well. An installation with a regular server risks the chance that any extra load results in a cascade of performance problems as jobs and processes quickly queue up and clog the whole system, making it almost unusable and requiring manual intervention by the administrators.

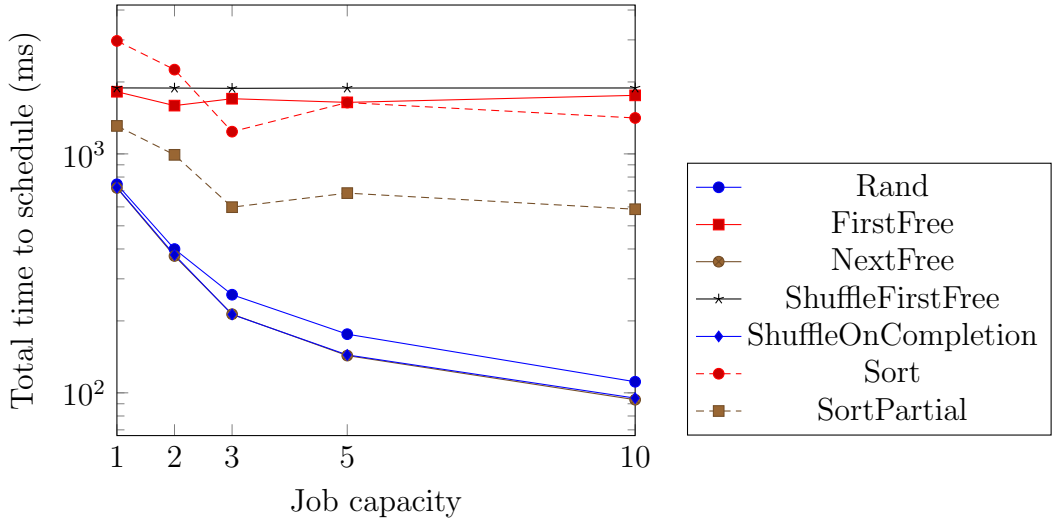

Figure 19: Algorithm test run 2: 1000 workers

From Figure 19, we see similar grouping behaviour as in Figure 18, but this time we can clearly see a performance drop when comparing units with 10 job capacity to ones with 1 job capacity.

Both NextFree and ShuffleOnCompletion slightly outperform the Rand-scheduler, showing that randomly picking workers from the list can be outperformed by a cleverly designed algorithm. The performance difference between the two is negligible, resulting in overlapping performance graphs. Difference between NextFree and ShuffleOnCompletion is almost non-existent, so we can both say that the extra load the shuffling introduces to ShuffleOnCompletion is negligible and that it does not really offer any significant improvement in performance in this case.

SortPartial has mostly broken out of the results of both FirstFree and Sort, outperforming both. The order of the workers starts to affect the performance of the scheduler more and more as total job capacity is greatly reduced.

ShuffleFirstFree has caught up with FirstFree and Sort, again giving steady performance between different job capacities. Sort has also caught up with FirstFree and even outperforming it in some cases, while giving the worst performance of all the algorithms with the job capacities of 1 and 2.

Test run 3: 100 workers

This run corresponds to an average installation which can range from 50 to 200 units. Performance here is not too crucial as the basic server can't easily fall into the same performance cascade the larger systems can due to the sheer decrease in scale.

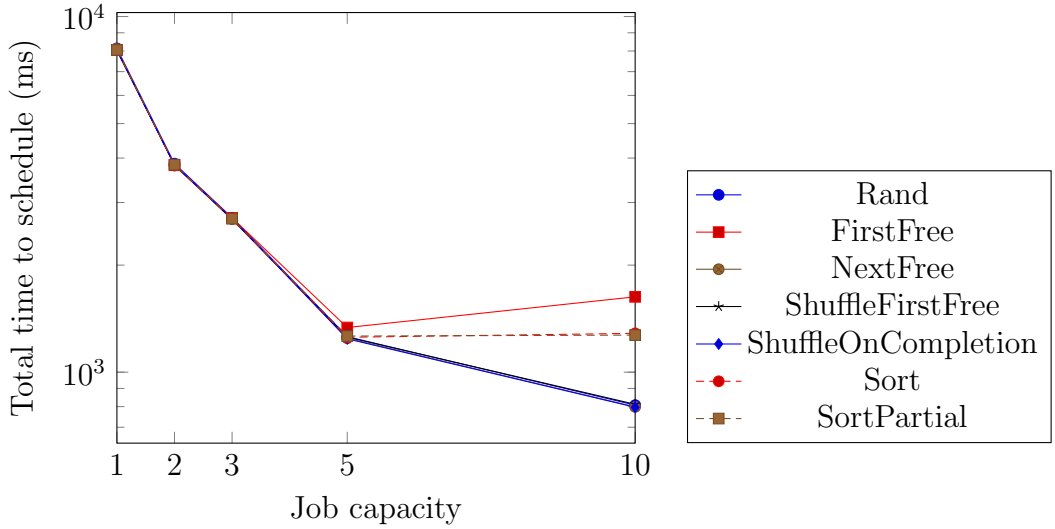

Figure 20: Algorithm test run 3: 100 workers

From Figure 20 we can see that all of the algorithms start to bundle up at job capacity 5 and completely overlapping from 3 onwards. This is the point where the algorithm does not weigh as much.

FirstFree seems to be the slowest at job capacity 10, while both Sort and SortPartial seem to give the same performance. This might be due to the edge both sorting schedulers achieve by having the most performant workers at the front and give similar performance as the number of sortable workers is so low SortPartial does not really benefit too much from the reduction of sortable units.

ShuffleFirstFree has convened with Rand, NextFree and ShuffleOnCompletion. This is most likely due to the small number of workers not requiring too much time to randomize and it is as fast to select a random worker by shuffling the whole list as it is just picking a random one from the list.

Test run 4: 10 workers

This test mostly corresponds to the currently smallest installations available, ranging from 1 to 50 units. These are mostly deployed to the cloud and require a very little processing power to run. Several of these can run on a single cloud instance. With these kinds of installations it could be feasible to have shared computing units serving all of them.

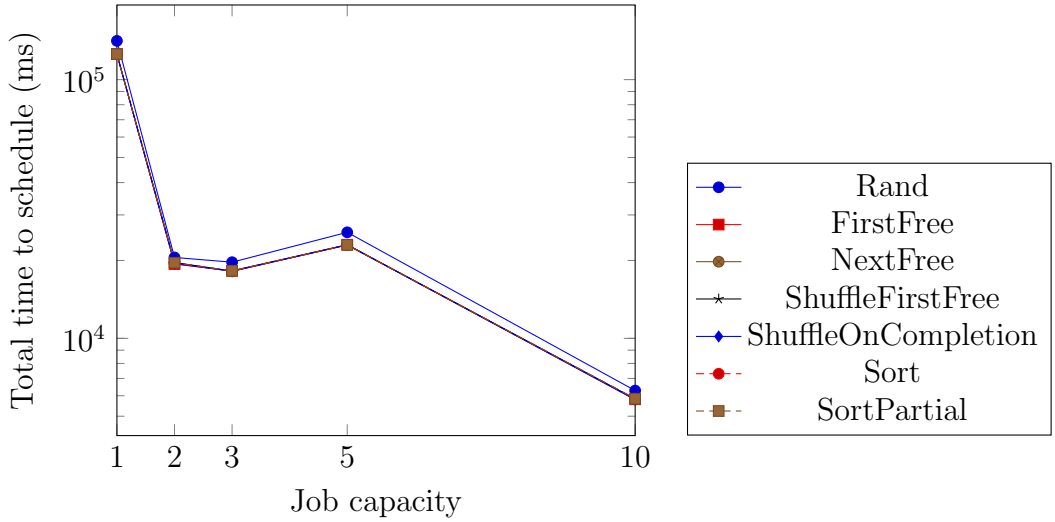

Figure 21: Algorithm test run 4: 10 workers

From the results in Figure 21, we can see that all of the algorithms have converged and that all of the usable algorithms outperform the Rand algorithm. This is due to the fact that the random algorithm has to generate a random number each time it tries to schedule a worker while all the other algorithms just loop over the list

until one of the workers has job capacity. At this stage, any one of algorithms would do just fine, as any extra operations any of them do to the list is so quick and inexpensive that it basically boils down to scanning the list and selecting the first free unit if any exists.

Test run 5: Semi-realistic simulation

This test is to see if the sorting algorithms can clearly benefit from the fact that they sort the fast 'offloading server' worker first and that the 'board' workers have such a large deviation in their computing time that they should almost always schedule all the available jobs to the most performant worker available.

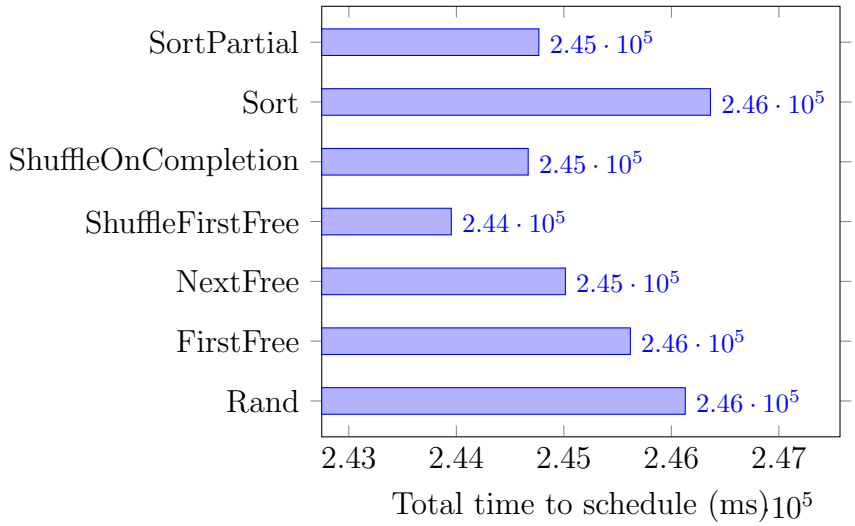

Figure 22: Algorithm test run 5: Semi-realistic simulation

Figure 22 shows us that while there are differences between all of the algorithms, no real difference is found. The difference between the fastest and the slowest algorithm is $\sim$ 2.4 seconds, which is $\sim$ 1% of the total benchmark time. No real advantage can be seen from either of the sorting algorithms and any differences between all the algorithms are most likely mostly due to the randomized nature of the worker job times.

7 CONCLUSIONS

This thesis introduced a distributed system for monitoring a parking area and the different components used currently in it. We proposed a system to distribute the computation of certain tasks computed by the boards between all the boards to reduce both the response time and processing load each unit has to perform. We partitioned the main video monitoring pipeline and found that two tasks, Task A and Task B, were taking almost all of the processing power each unit was using. These tasks were also using a very minor part of the input image to run, minimizing the amount of data we would have to send over the network. We also proposed a basic movement detection algorithm to be added to the pipeline before these tasks to minimize the amount of tasks spawned. New computation unit software was designed, containing the required libraries to support both Task A and Task B and created a basic protocol using gRPC for both computation unit discovery in the network and task sending. We designed a set of scheduling algorithms for load balancing which computation units would compute which tasks. We benchmarked them against a very basic random scheduler and found out that schedulers which remember where they last left off in the list of possible computing units performed as well as or better than the random scheduler. Sorting or randomizing the list turned out to be a rather heavy task and could not compete with the indexed schedulers. Between the randomized and sorted schedulers, only partitioning part of the list seems to be several times faster than sorting or randomizing the whole list. The partial sorter scheduler also outperformed the most basic scheduler, which returned the first free resource from the list. The basic functionality of these two was very similar to both always indexed the list from the beginning. Sorting the most performant resources first seems to have a positive performance impact, but not large enough to compete with the indexed schedulers.

All of the schedulers seemed to converge when the amount of resources dropped to 100 and their parallel job capacity to 5. From this point on it does seem that it does not really matter which scheduler is used, so we can rather safely say that in small installations the choice of a scheduler does not matter performance wise. The system seems to work rather well, but still need more live testing in production environments before it can properly be marked as working as intended. More testing is required to see that the fallback system works in the event of a network outage and that no existing installation breaks when this system is deployed.

Next step in this system is first to implement a working prototype and deploy it to a testing server. After this is done, more advanced metrics and tests can be collected and run. The most interesting of these is the potential response time improvement of

the sorting schedulers as while they cannot compete in pure scheduling performance with the indexed schedulers, sorting the resources could potentially save several milliseconds per task. Also it will be interesting to see if the basic movement detection algorithm can filter enough tasks from the pipeline so that an active parking lot does not run out of computation resources constantly.

The schedulers designed in this thesis were really basic and maybe not optimized for real use. Each scheduler was forced to use the same list of resources, while for example the sorting schedulers and the FirstFree scheduler would have maybe benefitted from having two separate lists of resources, one for ones with free capacity and another with no capacity. This was done to keep the complexity of the benchmark low just to see the general performance of each. Optimizing the algorithms further will be a large part of what's next. Also, while the scheduling case here does differ somewhat from CPU-schedulers and load balancing algorithms, some algorithms could be modified to work with or just drawn inspiration from for this use case with some additional work.

References

[1] N. A. Al Etawi. "A comparison between cluster, grid, and cloud computing". In: *International Journal of Computer Applications* 179.32 (2018), pp. 37–42.

[2] *Amazon EC2 — Amazon Web Services, Inc.* URL: `https://aws.amazon.com/ec2/` (visited on 03/30/2020).

[3] *Arch Linux*. URL: `https://www.archlinux.org/` (visited on 03/09/2020).

[4] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. *Operating systems: Three easy pieces*. Arpaci-Dusseau Books LLC, 2018.

[5] *AUR (en) - opencv-cuda*. URL: `https://aur.archlinux.org/packages/opencv-cuda/` (visited on 03/09/2020).

[6] *Average server response time for All websites 20k - 500k — Littledata Benchmarks*. URL: `https://www.littledata.io/average/server-response-time` (visited on 03/30/2020).

[7] *CDN Reliability & Redundancy — Cloudflare*. URL: `https://www.cloudflare.com/learning/cdn/cdn-load-balance-reliability/` (visited on 03/30/2020).

[8] *Cortex-A7 - Arm*. URL: `https://www.arm.com/products/silicon-ip-cpu/cortex-a/cortex-a7` (visited on 03/09/2020).

[9] *Documentation - The Go Programming Language*. URL: `https://golang.org/doc/` (visited on 03/09/2020).

[10] K. Erciyes. "Distributed mutual exclusion algorithms on a ring of clusters". In: *International Conference on Computational Science and Its Applications*. Springer. 2004, pp. 518–527.

[11] *Folding@home – Fighting disease with a world wide distributed super computer*. URL: `Foldingathome.org` (visited on 03/30/2020).

[12] E. J. Ghomi, A. M. Rahmani, and N. N. Qader. "Load-balancing algorithms in cloud computing: A survey". In: *Journal of Network and Computer Applications* 88 (2017), pp. 50–71.

[13] *Go Slices: usage and internals - The Go Blog*. URL: `https://blog.golang.org/go-slices-usage-and-internals` (visited on 03/09/2020).

[14] *gRPC*. URL: `https://grpc.io/` (visited on 03/09/2020).

[15] *gRPC — Concepts*. URL: `https://grpc.io/docs/guides/concepts/` (visited on 03/09/2020).

[16] *GStreamer: open source multimedia framework*. URL: `https://gstreamer.freedesktop.org/` (visited on 03/09/2020).

[17] C.-L. Hung, H.-h. Wang, and Y.-C. Hu. “Efficient load balancing algorithm for cloud computing network”. In: *International Conference on Information Science and Technology (IST 2012), April*. 2012, pp. 28–30.

[18] *Instruction Sets — Floating Point – Arm Developer*. URL: `https://developer.arm.com/architectures/instruction-sets/floating-point` (visited on 03/09/2020).

[19] *JPEG - JPEG*. URL: `https://jpeg.org/jpeg/index.html` (visited on 03/09/2020).

[20] S. Kaur, K. Kumar, J. Singh, and N. S. Ghumman. “Round-robin based load balancing in Software Defined Networking”. In: *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*. IEEE. 2015, pp. 2136–2139.

[21] L. Kirichenko, I. Ivanisenko, and T. Radivilova. “Dynamic load balancing algorithm of distributed systems”. In: *2016 13th International Conference on Modern Problems of Radio Engineering, Telecommunications and Computer Science (TCSET)*. IEEE. 2016, pp. 515–518.

[22] *Largest car park — Guinness World Records*. URL: `https://www.guinnessworldrecords.com/world-records/largest-car-park/` (visited on 03/30/2020).

[23] A. Lawrance and P. Lewis. “An exponential moving-average sequence and point process (EMA1)”. In: *Journal of Applied Probability* 14.1 (1977), pp. 98–113.

[24] D. B. LD and P. V. Krishna. “Honey bee behavior inspired load balancing of tasks in cloud computing environments”. In: *Applied Soft Computing* 13.5 (2013), pp. 2292–2303.

[25] M. Mesbahi and A. M. Rahmani. “Load balancing in cloud computing: a state of the art survey”. In: *International Journal of Modern Education and Computer Science* 8.3 (2016), p. 64.

[26] *MIPI Camera Serial Interface 2 (MIPI CSI-2)*. URL: `https://mipi.org/specifications/csi-2` (visited on 03/09/2020).

[27] K. Nishant, P. Sharma, V. Krishna, C. Gupta, K. P. Singh, R. Rastogi, et al. “Load balancing of nodes in cloud using ant colony optimization”. In: *2012 UKSim 14th international conference on computer modelling and simulation*. IEEE. 2012, pp. 3–8.

[28] K. A. Nuaimi, N. Mohamed, M. A. Nuaimi, and J. Al-Jaroodi. “A Survey of Load Balancing in Cloud Computing: Challenges and Algorithms”. In: *2012 Second Symposium on Network Cloud Computing and Applications*. 2012, pp. 137–142.

[29] *OpenCV: Color Space Conversions*. URL: https://docs.opencv.org/4.2.0/d8/d01/group__imgproc__color__conversions.html (visited on 03/09/2020).

[30] *OpenCV: Image file reading and writing*. URL: https://docs.opencv.org/4.2.0/d4/da8/group__imgcodecs.html (visited on 03/09/2020).

[31] *Oracle Solaris Cluster Oracle Technology Network*. URL: https://www.oracle.com/solaris/technologies/cluster-overview.html (visited on 03/30/2020).

[32] S. Penmatsa and G. S. Hura. "Performance Evaluation of System Optimal Load Balancing Schemes for Multi-User Job Distributed Systems". In: *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*. The Steering Committee of The World Congress in Computer Science, Computer ... 2017, pp. 68–73.

[33] *Protocol Buffers — Google Developers*. URL: https://developers.google.com/protocol-buffers (visited on 03/09/2020).

[34] C. Qu, R. N. Calheiros, and R. Buyya. "Mitigating impact of short-term overload on multi-cloud web applications through geographical load balancing". In: *concurrency and computation: practice and experience* 29.12 (2017), e4126.

[35] *runtime - The Go Programming Language*. URL: https://golang.org/pkg/runtime/ (visited on 03/09/2020).

[36] C. Scott, V. Brajkovic, G. Necula, A. Krishnamurthy, and S. Shenker. "Minimizing faulty executions of distributed systems". In: *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*. 2016, pp. 291–309.

[37] M. Sharma and S. Husain. "Analyzing the Difference of Cluster, Grid, Utility & Cloud Computing". In: *IOSR Journal of Computer Engineering* 19.1 (2017), pp. 55–60.

[38] D. Sood, H. Kour, and S. Kumar. "Survey of computing technologies: distributed, utility, cluster, grid and cloud computing". In: *Journal of Network Communications and Emerging Technologies (JNCET) www. jncet. org* 6.5 (2016).

[39] *sort - The Go Programming Language*. URL: https://golang.org/pkg/sort/ (visited on 04/25/2020).

[40] *src/sort/slice.go - The Go Programming Language*. URL: https://golang.org/src/sort/slice.go (visited on 04/25/2020).

[41] *src/sort/zfuncversion.go - The Go Programming Language*. URL: https://golang.org/src/sort/zfuncversion.go (visited on 04/25/2020).

[42] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. "Chord: A scalable peer-to-peer lookup service for internet applications". In: *ACM SIGCOMM Computer Communication Review* 31.4 (2001), pp. 149–160.

[43] I. Suzuki and T. Kasami. "A distributed mutual exclusion algorithm". In: *ACM Transactions on Computer Systems (TOCS)* 3.4 (1985), pp. 344–349.

[44] A. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2002. ISBN: 9780130888938. URL: https://books.google.fi/books?id=LYQpAQAAMAAJ.

[45] *The Difference Between RAM Speed and CAS Latency — Crucial.com*. URL: https://www.crucial.com/articles/about-memory/difference-between-speed-and-latency (visited on 03/30/2020).

[46] C. A. Varela and I. ebrary. *Programming distributed computing systems: a foundational approach*. English. Cambridge, Mass: MIT Press, 2013.

[47] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. "A note on distributed computing". In: *International Workshop on Mobile Object Systems*. Springer. 1996, pp. 49–64.

[48] *Yocto Project Overview and Concepts Manual*. URL: https://www.yoctoproject.org/docs/3.0.2/overview-manual/overview-manual.html (visited on 03/09/2020).